VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

SYSTÉM PRO VYHLEDÁVÁNÍ CHEMICKÝCH STRUKTUR

SYSTEM FOR SEARCHING OF CHEMICAL STRUCTURES

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. IVAN ŠEVČÍK**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. ZBYNĚK KŘIVKA, Ph.D.**
SUPERVISOR

BRNO 2018

Zadání diplomové práce/20667/2017/xsevci50

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2017/2018

Zadání diplomové práce

Řešitel: **Ševčík Ivan, Bc.**
Obor: Počítačová grafika a multimédia
Téma: **Systém pro vyhledávání chemických struktur**
System for Searching of Chemical Structures
Kategorie: Algoritmy a datové struktury

Pokyny:

1. Nastudujte problematiku hledání izomorfismu grafů a proveďte rešerši algoritmů a metod vhodných pro hledání identických struktur a podstruktur grafu. Prostudujte specifika algoritmů a metod při práci s chemickými strukturami/látkami.
2. Seznamte se s grafovými databázemi a zhodnoťte jejich přínos při řešení izomorfismu (pod)grafů při vyhledávání ve velkém množství dat.
3. Dle konzultací s vedoucím navrhněte a implementujte systém umožňující vyhledávání v databázi chemických látek.
4. Otestujte správnost a vyhodnoťte efektivnost implementace pokud možno na reálných datech.
5. Diskutujte dosažené výsledky a navrhněte budoucí vylepšení a rozšíření.

Literatura:

- Raymond, J.W. and Willett, P.: Maximum common subgraph isomorphism algorithms for the matching of chemical structures. *Journal of Computer-Aided Molecular Design*, 16: 521-533, 2002.
- Kotthoff, L., Mccreesh, C., and Solnon, C.: Portfolios of subgraph isomorphism algorithms. In *International Conference on Learning and Intelligent Optimization*, pp. 107-122, Springer, 2016.
- dle pokynů vedoucího

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Křivka Zbyněk, Ing., Ph.D.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 23. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Táto práca sa zaoberá problémom vyhľadávania štruktúr vo veľkých databázach chemických látok. Cieľom je návrh a implementácia efektívneho systému podporujúceho dva základné typy vyhľadávania, ktorými sú vyhľadanie identickej štruktúry a podštruktúrne vyhľadávanie. Okrem značného množstva záznamov v databázach komplikuje túto úlohu algoritmicky náročná, grafová reprezentácia chemických štruktúr. V rámci práce sú predstavené spôsoby, pomocou ktorých je možné tieto problémy úspešne riešiť. Súčasťou práce je aj vytvorenie webovej služby, ktorá vyhľadávanie sprístupňuje užívateľom.

Abstract

This thesis deals with the problem of searching of structures in large chemical compounds databases. The aim is to design and implement an efficient system that supports two basic types of search, which are identity and substructure search. This task is complicated not only by the large number of entries in databases but also by graph representation of chemical structures, for which many algorithms are hard to solve. The thesis will introduce concepts which will prove useful in solving these problems. A web service is also created as a part of the thesis in order to make the database searching available to the users.

Kľúčové slová

databáza chemických látok, systém pre vyhľadávanie, chemické štruktúry, izomorfizmus grafov a podgrafov, molekulárne odtlačky

Keywords

chemical compounds database, search engine, chemical structures, graph and subgraph isomorphism, molecular fingerprints

Citácia

ŠEVČÍK, Ivan. *Systém pro vyhledávání chemických struktur*. Brno, 2018. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Zbyněk Křivka, Ph.D.

Systém pro vyhledávání chemických struktur

Prehlásenie

Prehlasujem, že som túto prácu vypracoval samostatne pod vedením Ing. Zbyňka Křivku, Ph.D. Uviedol som všetky literárne zdroje a publikácie, z ktorých som čerpal.

.....
Ivan Ševčík
23. mája 2018

Poďakovanie

Rád by som poďakoval vedúcemu mojej diplomovej práce, Ing. Zbyňkovi Křivkovi, Ph.D. za rady pri jej vypracovaní a odborné vedenie. Taktiež som vďačný za konzultácie a usmernenie Jurajovi Lutišanovi a Robertovi Mistríkovi. V neposlednom rade ďakujem aj svojej rodine za všetku podporu počas celého štúdia a priateľom za udržiavanie dobrej nálady počas písania práce.

Obsah

Kapitola 1

Úvod

V posledných desaťročiach znamenal nástup informačných technológií premenu takmer všetkých odvetví. Podobne tomu bolo aj v oblasti chémie. V dnešnej dobe je napríklad možné virtuálne vytvárať a testovať nové liečivá alebo zisťovať vlastnosti látok predtým, než sú vôbec prvýkrát fyzicky vyrobené. Nemenej dôležité bolo aj zdigitalizovanie a zanesenie poznatkov o známych látkach do elektronických databáz. Vďaka sofistikovaným meracím prístrojom a rozsiahlym databázam je okrem iného možné testovať výrobky na prítomnosť toxických látok.

Práve databázy chemických látok, respektíve vyhľadávanie v nich, sú predmetom tejto práce. Potreba vyhľadávania vznikla prirodzene, jednak zo strany užívateľov, ktorí potrebujú nájsť určitú látku, ale aj zo strany informačných systémov, ktoré toho využívajú vo svojich algoritmoch. Táto práca sa venuje konkrétne vyhľadávaniu chemických látok na základe ich štruktúry, ktorá býva bežne reprezentovaná grafom. S narastajúcou veľkosťou databáz, ktoré v dnešnej dobe dosahujú až stovky miliónov záznamov, sa táto operácia stáva netriviálnou. Cieľom práce je preto vytvoriť čo najefektívnejší systém, ktorý umožní získavať záznamy z tak rozsiahlych databáz na základe rôznych kritérií v prijateľnom čase.

Systém predstavený v tejto práci je tvorený za účelom jeho ďalšieho použitia v pripravovanom projekte MolGate. Bude sa jednať o agregátor veľkého množstva online aj offline databáz chemických štruktúr, ktorý má slúžiť ako univerzálny vyhľadávač vo svete malých molekúl. Okrem toho bude možné integrovať MolGate so softvérovými balíkmi, a tým využiť štruktúrne informácie pre rôzne výpočty, simulácie, predikcie a ďalšie prípady použitia. Projekt vzniká z dôvodu, že chemické štruktúry a metadáta sú naprieč databázami uložené v rôznych formátoch, majú neúplné informácie a nie je v nich možné rýchlo a cielene vyhľadávať. MolGate preto niektoré dáta prenesie do vlastnej databázy, kde budú po spracovaní uložené a indexované. V prípadoch, kedy to bude možné, bude záznam obsahovať aj odkaz na pôvodný zdroj. V rámci agregovanej databázy bude potom možné vytvárať rôzne druhy dopytov, z ktorých niektoré v pôvodných databázach ani neboli možné.

Práca sa primárne venuje vytvoreniu riešenia z informatického hľadiska. Napriek tomu je pre pochopenie požiadaviek a správnu implementáciu nutná znalosť základov chémie atómov a molekúl. Kapitola 2, Chemické vlastnosti molekúl, je preto venovaná objasneniu chemických princípov a pojmov, ktoré sú v práci použité.

Následne sú v kapitole 3, Reprezentácie molekúl a algoritmy, predstavené spôsoby, akými je možné chemické štruktúry reprezentovať v informačných systémoch. Taktiež sú v tejto kapitole popísané vybrané koncepty a algoritmy, ktoré sa ukázali ako užitočné pri návrhu systému.

Kapitola 4, Analýza požiadaviek na systém, sa zaoberá predovšetkým objasnením motivácie pre vytvorenie systému a vytýčením požiadaviek, ktoré by mal výsledný systém spĺňať, ale aj analýzou niektorých existujúcich riešení.

V kapitole 5, Návrh riešenia, je predstavený návrh ľahko rozšíriteľného systému, ktorý podporuje požadované operácie vyhľadávania. Súčasťou je výber vhodných technológií, databázových riešení, voľba spôsobu uloženia dát v systéme a objasnenie mnohých ďalších rozhodnutí, ktoré bolo nutné pri návrhu systému uskutočniť. Niektoré významné aspekty implementácie navrhnutého systému sú následne popísané v kapitole 6, Popis implementácie.

Posledná kapitola 7, Merania a diskusia výsledkov, prezentuje výsledky rôznych meraní, medzi ktoré patrí porovnanie algoritmov pre hľadanie izomorfného podgrafu alebo meranie výkonnosti systému pre jednotlivé operácie vyhľadávania po tom, čo boli doň uložené milióny skutočných záznamov molekúl získaných z databázy PubChem.

Kapitola 2

Chemické vlastnosti molekúl

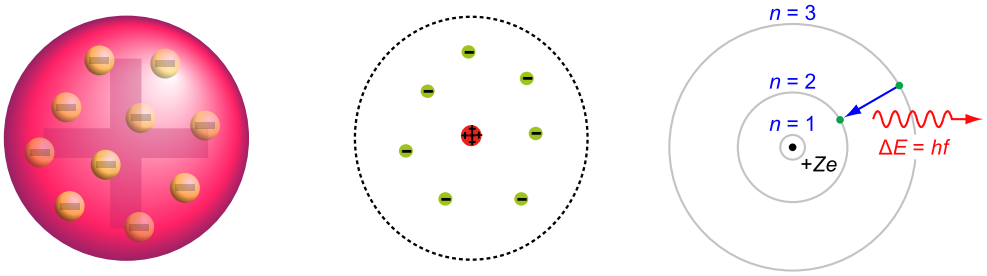
Ako bolo spomenuté v úvode, práca sa zaoberá vytvorením informačného systému pre vyhľadávanie chemických látok. Aby systém s týmito dátami pracoval správne a zmysluplne, je nutné všeobecné algoritmy prispôsobiť tak, aby brali v úvahu aj chemické vlastnosti látok. V nasledujúcom texte sú v krátkosti zhrnuté znalosti potrebné k pochopeniu týchto vlastností. V časti 2.1 sú najskôr predstavené atómy ako základné stavebné prvky látok. V časti 2.2 je potom popísané, akým spôsobom sa atómy spájajú do molekúl danej látky, a ako je možné molekuly zakresliť pomocou diagramov. Záverom sú vysvetlené niektoré ďalšie pojmy, ktoré sa v práci vyskytnú a majú dopad na fungovanie algoritmov.

2.1 Atóm

Jedným z prvých ľudí, ktorí prišli s myšlienkou, že všetko sa skladá z atómov, bol staroveký grécky filozof Demokritos. Jeho predstava bola, že existujú malé, pevné a ďalej nedeliteľné častice, z ktorých sa viditeľné objekty skladajú. V tej dobe sa však jeho teória neujala a trvalo niekoľko tisícročí, než John Dalton pri svojich experimentoch zistil, že pri reakcii vznikajú odlišné látky v závislosti od rôzneho hmotnostného pomeru vstupných látok, čiže reaktantov. Rozdiel v pomeroch bol pri jeho pokusoch často celým násobkom. Napríklad k vytvoreniu jednej látky bol vyžadovaný pomer uhlíka a kyslíka 1:2, ale pre vznik inej látky to bolo 1:4, čiže dvojnásobné množstvo kyslíka [28]. To ho nútilo prísť s teóriou, ktorá by tento jav vysvetľovala. V roku 1801 prišiel so svojou teóriou atómu, ktorá znie nasledovne:

- Hmota je zložená z malých častíc zvaných atómy (na počesť Demokrita).
- Atómy sú nedeliteľné a pri chemických reakciách sa iba preskupujú.
- Všetky atómy toho istého prvku majú rovnakú hmotnosť a ostatné vlastnosti.
- Atómy rôznych prvkov sa líšia hmotnosťou a vlastnosťami.
- Atómy sa môžu zlučovať do väčších celkov zvaných zlúčeniny. V danej zlúčenine sú atómy vždy v rovnakom celočíselnom pomere.

Skúmaniu pomerov sa ďalej venoval samostatný odbor chémie, stechiometria. Tvrdenie o nedeliteľnosti sa však neskôr ukázalo ako nepravdivé. Prvým krokom v tomto smere bolo objavenie elektrónu J.J. Thomsonom, ktorý zistil, že sa jedná o samostatnú časticu so záporným nábojom, ktorá musí mať menšiu hmotnosť než atóm [28]. Keďže sa však väčšina hmoty javí ako neutrálna, teda bez náboja, predpovedal aj existenciu kladne nabitej hmoty,



(a) Thomsonov model[1] (b) Rutherfordov model[2] (c) Bohrov model[3]

Obr. 2.1: Historický vývoj modelov atómu

ktorá by záporný náboj vyrovnala. Thomson si túto kladnú hmotu nepredstavoval ako samostatné častice, ale ako niečo, čo ním objavené častice v atóme obklopuje. Rutherford však pri svojich experimentoch zistil, že kladný náboj je skôr koncentrovaný na jednom mieste v strede atómu. Prišiel tak s novým modelom obsahujúcim atómové jadro. To sa podľa neho skladalo z kladne nabitých častíc, protónov, ale predpovedal aj existenciu neutrálne nabitých častíc, neutrónov, ktoré boli neskôr tiež experimentálne potvrdené. Pokusmi bolo ďalej zistené, že protóny a elektróny majú rovnako veľký, iba opačný náboj. Ak sa teda látka javí ako neutrálna, musí obsahovať rovnaký počet protónov a elektrónov. Práve počet protónov, nazývaný protónové alebo atómové číslo, charakterizuje o aký prvok sa jedná. Napríklad vodík obsahuje iba jeden protón, zatiaľ čo uhlík ich má 12. Súčet protónov a neutrónov sa nazýva nukleónové číslo. Ak majú atómy rovnaký počet protónov, ale líšia sa v počte neutrónov, jedná sa o izotopy rovnakého prvku. Izotopy majú podobné chemické vlastnosti, líšia sa zväčša rýchlosťou reagovania a niektoré z nich sú rádioaktívne.

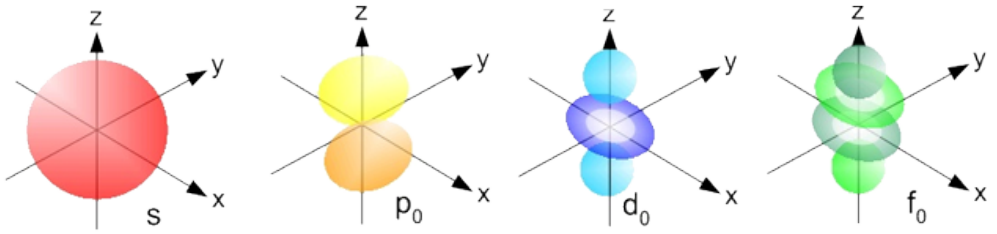
O niekoľko rokov po Rutherfordovi predstavil dánsky fyzik Niels Bohr svoj model atómu, v ktorom elektróny obiehajú v elektrónovom obale okolo jadra po dráhach, ktoré nazýval orbitami [28]. Dôležitým aspektom tohto modelu bolo, že jednotlivé dráhy mali rôzne energetické hladiny. Čím ďalej elektrón obiehal okolo jadra, tým väčšiu energiu mal. Aby sa elektrón presunul na vyššiu energetickú hladinu, musel energiu prijať a naopak pri presune na nižšiu hladinu energiu vyžiaril. Bohr tieto hladiny v svojom modely kvantifikoval, čiže určil niekoľko prípustných hladín. Ďalej určil maximálny počet elektrónov pre každú hladinu a definoval základný stav, pre ktorý platí, že všetky elektróny sú na najnižších možných energetických hladinách. Atóm v základnom stave má teda postupne zapĺňané orbity najbližšie k jadru, až kým je dosiahnutý maximálny počet elektrónov, kedy sa začne zapĺňať ďalšia hladina. Elektróny, ktoré sa nachádzajú na poslednej orbite, sa nazývajú valenčné elektróny a v podstatnej miere ovplyvňujú chemické vlastnosti atómu, ako je napríklad schopnosť reagovať s inými atómami [28]. Obrázok 2.1 zobrazuje postupný vývoj popísaných modelov.

Aktuálne najpresnejším modelom je kvantovo mechanický model atómu, ktorý vytvoril Erwin Schrödinger [15]. V ňom elektróny neobiehajú okolo jadra po kruhových orbitách, ale vyskytujú sa v oblastiach zvaných orbitály. Podľa Heisenbergovho princípu nie je možné predpovedať ľubovoľne presne polohu aj hybnosť elektrónu súčasne. Preto orbitály iba určujú oblasť, v ktorej je veľká pravdepodobnosť výskytu elektrónu s danou energiou. V kaž-

[1]Zdroj obrázka: https://commons.wikimedia.org/wiki/File:Plum_pudding_atom.svg
[2]Zdroj obrázka: https://commons.wikimedia.org/wiki/File:Rutherford_atomic_planetary_model.svg
[3]Zdroj obrázka: https://commons.wikimedia.org/wiki/File:Bohr-atom-PAR.svg

dom orbitáli môžu byť najviac dva elektróny. Každému elektrónu v tomto modely je možné priradiť štyri kvantové čísla:

- Hlavné kvantové číslo, označované písmenom n, udáva energetickú hladinu podobne ako v Bohrovom modely.
- Vedľajšie kvantové číslo l udáva tvar orbitálu. Najbežnejšie sú štyri orbitály s ($l = 0$), p ($l = 1$), d ($l = 2$) a f ($l = 3$), ale prípustné sú hodnoty až do $n - 1$.
- Magnetické kvantové číslo m určuje priestorovú orientáciu orbitálu. Prípustné sú hodnoty z intervalu $[-l, l]$.
- Spin s špecifikuje, o ktorý z dvoch elektrónov v orbitáli sa jedná. Hodnota je buď $1/2$, alebo $-1/2$.



Obr. 2.2: Základné orbitály[1]

Príklad niektorých orbitálov je na obrázku 2.2. K tomu, aby bol prvok chemicky stály, má tendenciu zaplniť svoje s a p orbitály. V s orbitáli sa môžu nachádzať dva elektróny a v troch možných p orbitáloch ďalších šesť elektrónov. Spolu týchto osem elektrónov tvorí takzvaný elektrónový oktet. Prvky, ktoré majú tieto orbitály zaplnené prirodzene, napríklad vzácne plyny neón alebo argón, sú veľmi stále a nereagujú s inými prvkami. Väčšina ostatných prvkov, ako bude uvedené v časti 2.2, vytvára často zlúčeniny, vďaka ktorým môžu taktiež dosiahnuť elektrónový oktet na valenčnej vrstve [17].

2.2 Molekula

Ako bolo uvedené v predchádzajúcej časti 2.1, atómy sa môžu zlučovať a vytvárajú tak zlúčeniny. Rozdiel medzi molekulou a zlúčeninou je len v tom, že zlúčeninu musia tvoriť aspoň dva odlišné prvky. Napríklad ozón O_3 je molekula, ale nie zlúčenina, zatiaľ čo chlorid vápenatý $CaCl_2$ je oboje [28]. V práci sa však ďalej tento rozdiel nebude uvažovať. Atómy sa zlučujú do molekúl za účelom zníženia celkovej energie. Z hľadiska energie je napríklad výhodnejšie pre atómy vodíka vytvoriť molekulu H_2, ako existovať samostatne. K zlúčeniu atómov dochádza pomocou väzieb, ktoré sú výsledkom vzájomného pôsobenia nabitých častíc. Tri základné typy väzieb sú:

- Iónová – vzniká medzi kladne nabitými katiónmi a záporne nabitými aniónmi na základe elektrickej príťažlivosti opačných nábojov.

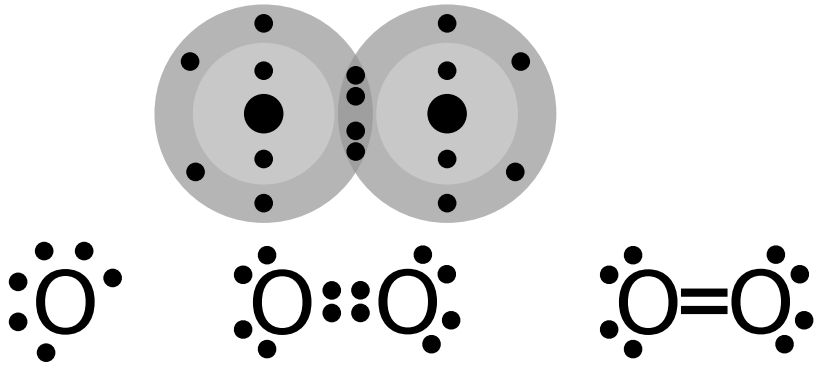
[1] Zdroj obrázka: https://chem.libretexts.org/Core/Physical_and_Theoretical_Chemistry/Quantum_Mechanics/09._The_Hydrogen_Atom/Atomic_Theory/Electrons_in_Atoms/Electronic_Orbitals

- Kovalentná – vzniká prekrytím a splynutím orbitálov obsahujúcich nespárovaný elektrón. Toto splynutie je možné chápať ako zdieľanie elektrónov zúčastňujúcich sa na väzbe medzi oboma atómami.
- Kovová – vzniká medzi atómami kovu, kedy dochádza k odtrhnutiu elektrónov, ktoré sa potom môžu voľne pohybovať v okolí atómových jadier. Z tejto konfigurácie vyplýva veľa vlastností kovov, napríklad vysoká pevnosť, vodivosť alebo ich lesk.

Typ väzby, ktorá medzi atómami vznikne, závisí od rozdielu elektronegativít. Elektronegativita je vlastnosť atómu odvodená chemikom Paulingom, ktorá vyjadruje schopnosť priťahovať väzbové elektróny [15]. Ako príklad iónovej väzby je možné uviesť chlorid sodný $Na^{+}Cl^{-}$. Atóm sodíka je veľmi reaktívny a postačuje malé množstvo energie pre odtrhnutie jediného elektrónu na poslednej valenčnej vrstve. Jeho odtrhnutím atóm dosiahne elektrónový oktet a stane sa stabilnejším. Naopak atómu chlóru jeden elektrón do oktetu chýba a je preň výhodné voľný elektrón prijať. Tým vzniknú dva opačne nabité ióny, ktoré spolu vytvoria molekulu. V práci sa však takmer výhradne stretneme s kovalentnými väzbami, ktoré budú v nasledujúcom texte popísané podrobnejšie. K tomu je vhodné najskôr predstaviť Lewisovu štruktúru.

2.2.1 Lewisova štruktúra

Lewisova štruktúra je zápis molekuly, v ktorom sú vyobrazené jednotlivé atómy, ich valenčné elektróny a väzby medzi atómami [15]. Základom je symbol pre každý atóm pozostávajúci zo skratky chemického prvku obklopeného bodkami, ktoré reprezentujú valenčné elektróny. Dvojicu valenčných elektrónov, ktoré sa podieľajú na väzbe, je možné prekresliť ako jednu čiaru, ktorá má význam jednoduchej väzby. Ak sa na väzbe podieľa viacero dvojíc, prekreslia sa ako súbežné čiary. Vzniká tak dvojitá, prípadne trojitá väzba. Obrázok 2.3 zobrazuje symbol pre jeden atóm kyslíka a spôsob, akým dva atómy kyslíka vytvoria jednu molekulu O_2 s dvojitou väzbou. Prostredná časť navyše zobrazuje Bohrov model pre túto molekulu.

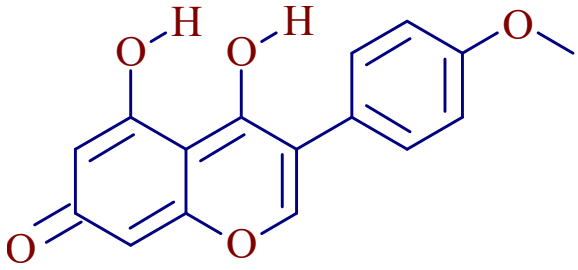


Obr. 2.3: Lewisova štruktúra pre molekulu O_2

Výhodou tohto zápisu je, že umožňuje vizualizovať elektrónové oktety, ktoré pri väzbách vznikajú. Ako je vidieť na obrázku 2.3, každý atóm kyslíka má štyri samostatné valenčné elektróny a ďalšie štyri sú zdieľané medzi oboma atómami, čím sa vytvoril oktet.

2.2.2 Štruktúrny chemický vzorec

Lewisova štruktúra je len jeden z mnohých grafických zápisov zlúčenín. V chémii je možné stretnúť sa s mnohými ďalšími v závislosti na tom, aká časť alebo vlastnosť molekuly je

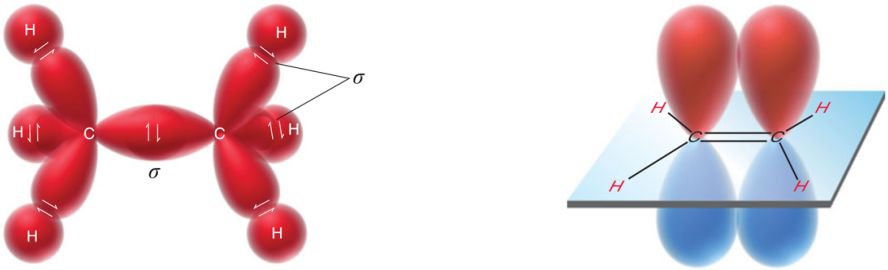


Obr. 2.4: Štruktúrny vzorec organickej molekuly

významná. Pri organických zlúčeninách sa často využíva štruktúrny chemický vzorec (angl. skeletal formula), v ktorom sa zväčša nezapisujú atómy uhlíka ani ich väzby s atómami vodíka. Je to kvôli tomu, že molekuly v organickej chémii sú z veľkej časti zložené práve z týchto dvoch prvkov, a ich zápis by bol teda zbytočne prácny. Obrázok 2.4 uvádza ako príklad štruktúrny vzorec pomerne malej organickej molekuly.

2.2.3 Kovalentné väzby

Pri kovalentných väzbách môže dôjsť k dvom rôznym prekrytiam orbitálov. Prekrytím orbitálov v ose spájajúcej stredy atómov vznikne σ väzba. Nemusí sa pritom jednať o rovnaké orbitály, k prekrytiu môže dôjsť napríklad medzi dvoma s orbitálmi, ale aj medzi s a p orbitálom. Na obrázku 2.5a je možné vidieť σ väzby medzi rôznymi orbitálmi. π väzba vzniká medzi súbežnými orbitálmi, najčastejšie p alebo d. Kvôli menšiemu prekrytiu orbitálov, ako je možné vidieť na obrázku 2.5b, sú π väzby slabšie ako σ [17].

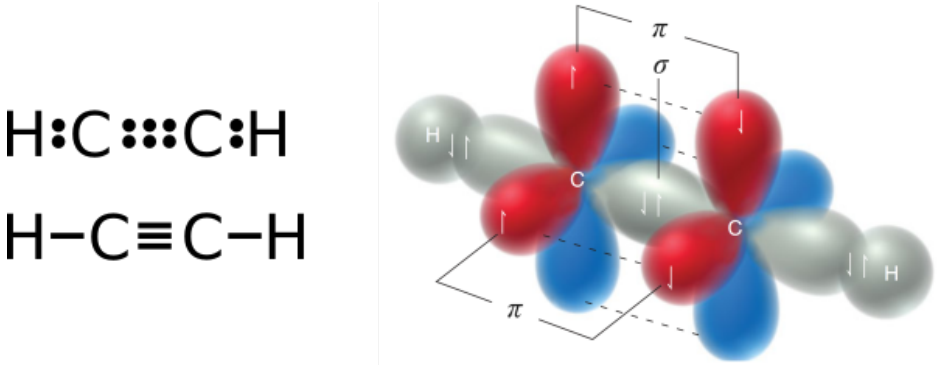


(a) σ väzba (b) π väzba

Obr. 2.5: Dva druhy prekrytí pri kovalentných väzbách [17]

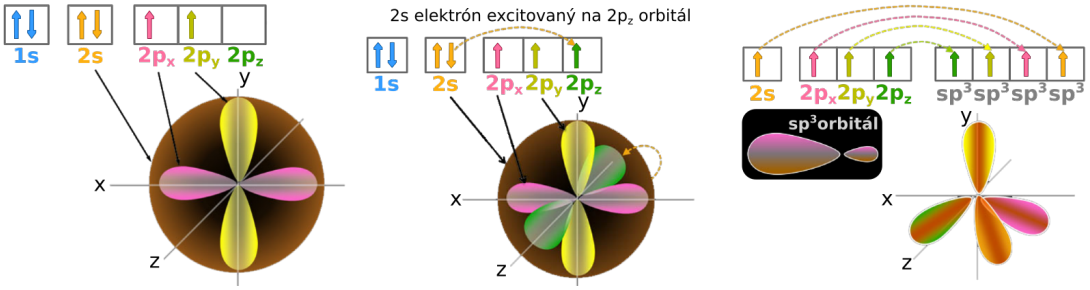
Pri popise Lewisovej štruktúry v časti 2.2.1 bolo spomenuté, že podľa počtu zdieľaných elektrónov vzniká jednoduchá, dvojitá alebo trojitá väzba. Tie priamo odpovedajú počtu prekrytých orbitálov. Jednoduchá väzba odpovedá σ väzbe, dvojitú väzbu tvorí jedna σ a jedna π väzba a trojitá väzba je tvorená jednou σ a dvoma π väzbami [17]. Na obrázku 2.6 je možné vidieť trojitú väzbu, ktorá je medzi atómami uhlíka v acetyléne. Čím väčšia je násobnosť väzby, tým väčšia energia je potrebná k oddeleniu atómov a tým menšia je vzdialenosť medzi atómami.

Pri niektorých atómoch sú bežné väzby, ktoré sa nedajú vysvetliť pomocou základných orbitálov. Napríklad atóm uhlíka `C` má nespárované elektróny iba v dvoch p orbitáloch, z čoho by sa dalo usúdiť, že bude môcť vytvoriť iba dve väzby. Napriek tomu zvykne atóm uhlíka zdieľať štyri valenčné elektróny, vďaka čomu môže vytvoriť elektrónový oktet. Tento jav sa pokúsil vysvetliť Pauling pomocou hybridizácie orbitálov [17]. Pri nej



Obr. 2.6: Prekrytie orbitálov v rámci trojitej väzby v acetyléne a znázornenie pomocou Lewisovej štruktúry [17]

dochádza k splynutiu a spriemerovaniu viacerých orbitálov do nových foriem, ktoré bývajú pomenované podľa pôvodných orbitálov, z ktorých sa skladajú. Napríklad splynutím jedného s a troch p orbitálov vznikajú štyri sp^3 orbitály tak, ako je uvedené na obrázku 2.7. Práve k tomu dochádza v molekule etánu na obrázku 2.5a, v ktorej môžu atómy uhlíka vytvoriť štyri väzby. Iným prípadom hybridizácie je sp^2, kedy dochádza k splynutiu s orbitálu a iba dvoch p orbitálov. Tretí p orbitál tak zostáva zachovaný. Tento orbitál sa potom môže podieľať na vytvorení dvojitej väzby tak, ako je to v prípade eténu na obrázku 2.5b.

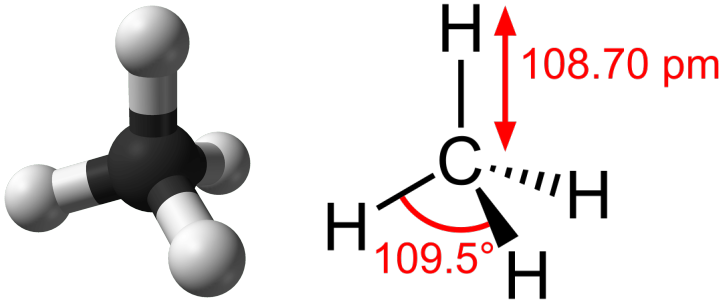


Obr. 2.7: Vznik hybridných sp^3 orbitálov[1]

2.2.4 Geometria molekúl

Molekuly a atómy sú objekty v trojrozmernom priestore. Tvar molekuly má podstatný dopad na jej vlastnosti, ako je napríklad reaktivita, farba alebo biologická aktivita. Stereochémia je oblasť chémie skúmajúca priestorové usporiadanie atómov v zlúčeninách. Molekuly sú však často zakreslené ako dvojrozmerné diagramy. Tento zápis je väčšinou postačujúci, pretože atómy je možné umiestniť do jednej roviny. Existujú ale konfigurácie atómov, pri ktorých už takú rovinu nie je možné nájsť. Z tohto dôvodu sa zaviedla klinová notácia (angl. wedge-dash notation) [15], v ktorej je priestorovosť vyjadrená špeciálnym tvarom väzieb. Vyplnený klin značí väzbu, ktorá smeruje z roviny nákresu von k pozorovateľovi. Prerušovaný klin alebo niekedy iba čiara sa potom použijú pre väzbu, ktorá je smerom od pozorovateľa.

[1]Zdroj obrázka: http://www.chemistryland.com/CHM151S/09-CovalentBonds/Covalent.html



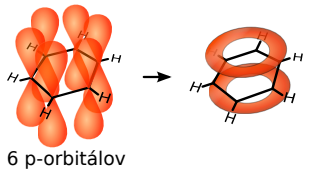
(a) 3D model molekuly[1] (b) 2D diagram[2]

Obr. 2.8: Priestorová reprezentácia molekuly metánu

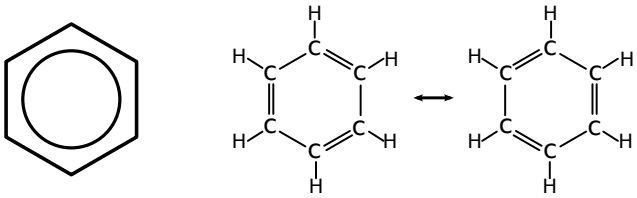
Geometriu molekuly je v dnešnej dobe možné určiť na základe teórie odpudzovania valenčných elektrónových párov (VSEPR) [15], ktorej popis je nad rámec tejto práce. Na základe tejto teórie bolo odvodených niekoľko bežných priestorových konfigurácií, napríklad štvorsten v prípade hybridizácie sp^3. Na obrázku 2.8 je možné vidieť túto konfiguráciu pre jednoduchú molekulu metánu CH_4 s odpovedajúcim 2D diagramom.

2.2.5 Ďalšie pojmy

Aromaticita je pojem charakterizujúci cyklické rovinné molekuly, v ktorých p orbitály navzájom splývajú [17]. Vďaka tomuto prepojeniu orbitálov sa jedná o veľmi stále molekuly. Najčastejšie sa jedná o uhľovodíky, teda zlúčeniny uhlíkov a vodíkov. Na obrázku 2.9 je možné vidieť orbitály pre benzén, známy prípad aromatickej molekuly. Väzby v aromatických molekulách sa zvyknú zakresľovať ako striedajúce sa jednoduché a dvojité väzby. Keďže sa ale jedná o vzájomné splynutie všetkých p orbitálov, sú zápisy na obrázku 2.10 ekvivalentné. Diagramy, medzi ktorými je na obrázku šípka, sa zvyknú nazývať **rezonančné formy**. Týmto pojmom sa všeobecne označujú rôzne zakreslenia tej istej látky.



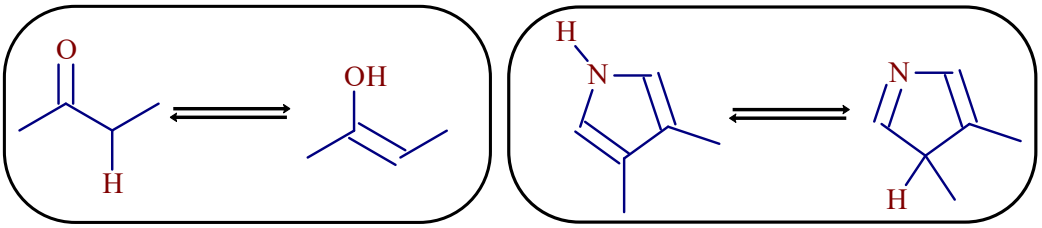
Obr. 2.9: Orbitály v benzéne[3]



Obr. 2.10: Rôzne značenia benzénu[4]

Tautoméry sú molekuly s rovnakým chemickým vzorcom, ale inou štruktúrou, ktoré sa môžu jednoducho meniť na iné formy reakciou zvanou tautomerizácia [17]. Bežnou takou reakciou je migrácia protónu, čiže vodíka H^+, pozdĺž série striedajúcich sa jednoduchých a dvojitých väzieb. Migrujúci vodík sa tiež zvykne označovať ako mobilný.

[1] Zdroj obrázka: https://commons.wikimedia.org/wiki/File:Methane-CRC-MW-3D-balls.png
[2] Zdroj obrázka: https://commons.wikimedia.org/wiki/File:Methane-2D-dimensions.svg
[3] Zdroj obrázka: https://commons.wikimedia.org/wiki/File:Benzene_Orbitals.svg
[4] Zdroj obrázka: https://commons.wikimedia.org/wiki/File:Benzene_Representations.svg



Obr. 2.11: Príklady tautomérov

Tautoméry sa v prírode vyskytujú v určitom pomere. Niektoré formy sú teda preferovanejšie než iné a tautomerizácia je prostriedok, ktorým sa táto rovnováha nadobudne. Na obrázku 2.11 sú ako príklad uvedené dva páry tautomérov.

Vodíkový mostík je druh interakcie medzi molekulami, prípadne v rámci jednej molekuly. Vzniká na základe elektrostatickej príťažlivosti medzi atómom vodíka, ktorý je kovalentne viazaný na vysoko elektronegatívny prvok a iným atómom s vysokou elektronegativitou. Jedná sa teda o akýsi druh väzby, ktorá je slabšia ako kovalentná, ale silnejšia než ostatné medzimolekulové sily.

Kapitola 3

Reprezentácie molekúl a algoritmy

Molekuly je nutné v informačných systémoch určitým spôsobom reprezentovať. Kapitola 2 predstavila niekoľko z mnohých popisov používaných v chémii. Cieľom tejto kapitoly je predstaviť reprezentácie, ktoré sú naopak vhodné z pohľadu informatiky. Kapitola sa tiež venuje popisu algoritmov a konceptov týkajúcich sa predovšetkým grafovej reprezentácie, ktoré budú neskôr využité pri návrhu riešenia.

Štruktúrny vzorec je pre väčšinu aplikácii vhodnou formou popisu, pretože poskytuje dostatočnú mieru konkrétnosti a zároveň predstavuje vhodnú mieru abstrakcie. Je z neho totiž možné určiť napríklad vzájomnú polohu atómov, jednotlivé prvky a väzby, ale nepopisuje molekulu až na kvantovej úrovni. Ďalšou výhodou je možnosť intuitívnej reprezentácie pomocou grafu, ktorý je v informatike bežnou dátovou štruktúrou. Tejto forme popisu a taktiež predstaveniu základov teórie grafov sa venuje sekcia 3.1.

Štruktúrny vzorec však nie je vhodný vo všetkých prípadoch. Tým, že sa jedná o grafický zápis, nie je ho možné jednoducho zdieľať napríklad v odkaze. Taktiež algoritmy pre prácu s grafmi sú náročné na výpočtové prostriedky, preto je snaha využiť aj iné spôsoby popisu. V sekcii 3.2 sú predstavené textové reprezentácie, ktoré boli vytvorené za účelom popisu molekúl.

3.1 Grafová reprezentácia chemických štruktúr

V tejto sekcii sú uvedené základy teórie grafov a súvislosť so štruktúrnym vzorcom chemických látok. Z tohto popisu molekúl budú neskôr vychádzať aj dátové štruktúry, ktoré ukladajú štruktúrne dáta pre molekuly v rámci implementovaného systému.

3.1.1 Definícia grafu

Keďže sa v texte tejto práce bude často používať termín graf a ďalšie termíny s ním spojené, je vhodné ich v krátkosti zadefinovať. Tieto definície sú prevzaté prevažne z knihy [11]. Orientovaný graf je usporiadaná trojica $G = (V, E, \varepsilon)$. V je neprázdna konečná množina, ktorej prvky sa nazývajú *vrcholy*. Ďalej E je konečná množina, ktorej prvky sa nazývajú *orientované hrany*, a ε je zobrazenie $E \to V^2$, ktoré každej hrane $e \in E$ priraďuje usporiadanú dvojicu vrcholov (x, y). Zápis značí, že hrana *vedie* z vrcholu x do vrcholu y. Tiež je možné povedať, že hrana tieto vrcholy *spája*. V prípade neorientovaného grafu je E množina *neorientovaných hrán* a zobrazenie ε priraďuje každej hrane neusporiadanú dvojicu vrcholov $[x, y]$, teda $[x, y] = [y, x]$. V oboch prípadoch sa vrcholy x a y nazývajú *krajnými vrcholmi* hrany e a navzájom sú tieto vrcholy *susedné*. V práci sa neuvažuje prípad, kedy

$x = y$, čiže v grafoch sa nevyskytujú *slučky*. Pre určitý vrchol sa počet susedných vrcholov označuje ako *stupeň* vrcholu.

Sled je postupnosť vrcholov a hrán $v_0, e_1, v_1, e_2, v_2, \ldots, e_k, v_k$, ak pre každé e_i platí $\varepsilon(e_i) = (v_{i-1}, v_i)$, resp. $\varepsilon(e_i) = [v_{i-1}, v_i]$. Sled *vedie* z vrcholu v_0 do vrcholu v_k a tieto vrcholy sa nazývajú *krajné*. Sled, v ktorom sa neopakuje žiadny vrchol, sa nazýva *cesta*. Ak $v_0 = v_k$, jedná sa o *uzavretý* sled. *Uzavretá cesta* je uzavretý sled, v ktorom sa okrem krajných vrcholov neopakuje žiadny vrchol ani hrana. Pre uzavretú cestu sa v orientovaných grafoch používa názov *cyklus* a v neorientovaných *kružnica*.

Vrcholy u a v spolu *súvisia*, ak existuje cesta z u do v. Ak každé dva vrcholy súvisia, tak sa graf G nazýva *súvislý*. Graf G' je *podgrafom* grafu G, ak vznikne z grafu G vynechaním niektorých, prípadne žiadnych vrcholov a hrán. Dôležité však je, aby zostala zachovaná definícia grafu, teda napríklad aby každá hrana mala oba koncové vrcholy. *Komponenta* grafu G je každý maximálny podgraf H grafu G, ktorý je súvislý.

3.1.2 Štruktúra molekuly ako graf

Štruktúrny vzorec svojou grafickou formou pripomína grafovú štruktúru. Toho si všimol Sylvester už v roku 1878, krátko po samotnom zavedení tejto formy popisu molekúl. Vo svojom článku skúma vzťah medzi týmito dvoma konceptmi a prvý krát používa v kontexte chemických štruktúr termín graf v dnešnom zmysle teórie grafov [25]. V tejto práci sa ale bude vychádzať prevažne z grafickej podobnosti pre jej intuitívnosť.

Tak ako sa grafová štruktúra skladá z vrcholov a hrán, ktoré vrcholy spájajú, aj chemické štruktúry sa skladajú z atómov, ktoré sú v molekule spojené pomocou chemických väzieb. Na väzbe sa podieľajú oba atómy súčasne, a preto nemá zmysel hovoriť o orientácii. Výnimkou je stereochémia, kde smer zakreslenia klinu značí priestorové usporiadanie molekuly, ale tento prípad bude možné riešiť bez nutnosti použitia orientovaných hrán. Štruktúrny vzorec je teda možné považovať za neorientovaný graf. V rámci grafovej štruktúry, tak ako bola definovaná, však nie je možné popísať okrem spojitosti chemickej štruktúry žiadne jej ďalšie aspekty. Predovšetkým každý atóm je určitý prvok, ale aj atómy rovnakého prvku sa môžu líšiť nábojom alebo počtom neutrónov. Taktiež väzby majú rôznu násobnosť a stereochémiu. Je preto potrebné, aby každý vrchol a hrana mohli mať množinu vlastností s konkrétnou hodnotou. Pre vrcholy je potrebné mať možnosť špecifikovať prvok, izotop, náboj a počet radikálnych elektrónov. U hrán je nutné pridať typ väzby a stereo orientáciu. Takto rozšírená grafová štruktúra už je schopná plne reprezentovať chemickú štruktúru.

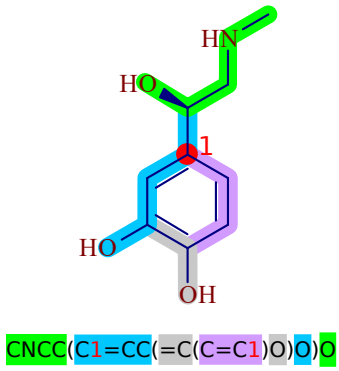
3.2 Textová reprezentácia molekúl

Najzákladnejšou textovou reprezentáciou molekuly v chémii je jej molekulárny vzorec. Ten nesie iba informáciu o počte jednotlivých prvkov v molekule, aj keď existujú pravidlá v akom poradí sa jednotlivé prvky vo vzorci vyskytnú. Ako príklad je možné uviesť molekulárny vzorec vody H_2O alebo zložitejšej molekuly glukózy $C_6H_{12}O_6$. Problém tejto reprezentácie je, že až na najjednoduchšie a najbežnejšie sa vyskytujúce látky je nejednoznačná. Inými slovami, existuje množstvo látok s rovnakým molekulárnym vzorcom. V tejto časti budú predstavené ďalšie dve textové reprezentácie, ktoré popisujú molekulu jednoznačnejšie, na vyššej úrovni detailu a sú vhodné pre spracovanie počítačom.

3.2.1 SMILES a SMARTS

Simplified Molecular Input Line Entry System (SMILES) je textový reťazec pre zápis a reprezentáciu molekúl a chemických reakcií. SMILES popisuje rovnaké informácie, ktoré je možné nájsť v rozšírenej grafovej reprezentácii chemických štruktúr, ale jeho výhodou je, že sa jedná o druh textového zápisu na rozdiel od dátovej štruktúry. SMILES je založený na pomerne jednoduchom jazyku, ktorý je tvorený niekoľkými gramatickými pravidlami [10].

Základné konštrukcie v SMILES sú atómy, väzby, vetvenia a cykly. Atómy sa zapisujú do hranatých zátvoriek, v ktorých je uvedená skratka prvku, počet naviazaných vodíkov a náboj. Napríklad `[OH3+]` značí kladne nabitý atóm kyslíka s troma atómami vodíka. Pre najbežnejšie organické prvky v základnej konfigurácii postačuje uviesť skratku prvku, napríklad `C` je vo význame `[CH4]`. Medzi každými dvoma atómami v zápise existuje väzba, pričom jednoduché väzby nie je potrebné uvádzať. Jednoduchá väzba sa značí `-`, dvojitá `=`, trojitá `#` a aromatická `:`. Tieto pravidlá umožňujú zapísať jeden reťazec atómov. Pre popis vetvenia štruktúry bola zavedená notácia používajúca zátvorky `()`. V mieste, kde sa molekula rozvetvuje, sa do zátvoriek uvedie zápis bočnej vetvy a ďalej sa pokračuje v pôvodnej vetve. Zátvorky je možné do seba zanoriť. Posledné pravidlo umožňuje popísať cykly v chemických štruktúrach. V určitom mieste sa cyklus rozdelí a toto miesto sa označí číslom. Toto číslo sa v reťazci znova vyskytne na mieste, kde má dôjsť k opätovnému napojeniu. Najľahšie je možné tieto pravidlá demonštrovať na praktickom príklade. Obrázok 3.1 zobrazuje molekulu adrenalínu a uvádza jej SMILES identifikátor. Jednotlivé vetvy sú farebne odlíšené. Tiež je zvýraznený bod rozdelenia cyklu, v tomto prípade benzénového jadra.



Obr. 3.1: Adrenalín a jeho štandardný SMILES identifikátor

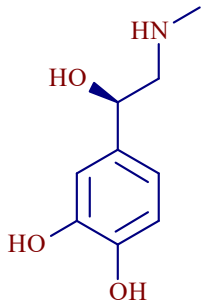
SMILES má hneď niekoľko možných použití. Jedným z nich je uloženie chemických štruktúr v databáze, keďže sa jedná o úsporný zápis, ktorý je navyše možné ďalej komprimovať. Taktiež je možné vytvoriť štandardný SMILES, ktorý je potom možné použiť ako unikátny identifikátor danej molekuly. V praxi sa však toto použitie ukázalo ako problematické kvôli odlišnostiam medzi implementáciami, pretože SMILES vznikol ako súčasť komerčného softvéru a pôvodné zdrojové kódy neboli dostupné. Toto bolo jedným z podnetov pre vznik ďalšieho identifikátora, ktorý je popísaný v nasledujúcej sekcii 3.2.2.

SMILES tvorí základ pre ďalší jazyk, SMARTS. Ten umožňuje na rozdiel od priameho popisu chemickej štruktúry popísať rôzne vzory, ktoré pripomínajú regulárny výraz určený pre chemické štruktúry. Pomocou symbolu `*` je napríklad možné značiť atóm akéhokoľvek prvku a symbol `~` vo význame väzby značí, že sa môže jednať o ľubovoľný typ. Jazyk ďalej pridáva logické operátory, takže je možné zapísať napríklad `[C,O]` vo význame atóm uhlíka *alebo* kyslíka. Plnú špecifikáciu jazyka je možné nájsť na stránkach firmy Daylight, ktorá

ju vytvorila [9]. Keďže sa jedná o pomerne jednoduchý jazyk, môže slúžiť ako alternatívna forma vstupu, pomocou ktorej užívateľ zapíše vzor, ktorý požaduje vyhľadať v databáze. Tento jazyk je ale možné nájsť použitý aj priamo v implementácii niektorých častí chemoinformatického softvéru, napríklad pri vytváraní molekulárnych odtlačkov, ktoré budú predstavené v sekcii 3.3.

3.2.2 InChI

IUPAC International Chemical Identifier (InChI) je textový reťazec pozostávajúci čisto z ASCII znakov, ktorý umožňuje unikátnu reprezentáciu chemických látok. Jedná sa o identifikátor, ktorý je generovaný na základe poskytnutej chemickej štruktúry [16]. Ako netriviálny príklad je možné uviesť adrenalín na obrázku 3.2. InChI umožňuje popisovať tiež molekuly, ktoré majú v grafovej reprezentácii viacero komponent.



InChI=1S/C9H13NO3/c1-10-5-9(13)6-2-3-7(11)8(12)4-6/h2-4,9-13H,5H2,1H3/t9-/m0/s1

Obr. 3.2: Adrenalín a zodpovedajúci InChI identifikátor

Keďže sa má jednať o akýsi digitálny podpis danej látky, je nutné, aby boli zaručené dve vlastnosti. Prvou je, že chemicky odlišné látky budú mať rozdielne identifikátory. Tou druhou je, že konkrétna látka musí mať iba jeden identifikátor, bez ohľadu na to, ako bola nakreslená alebo inak vytvorená. Špecifikácia InChI sa neustále vyvíja a má niekoľko verzií. V čase písania tejto práce bola najaktuálnejšia verzia `1.05`, ktorá bola vydaná v roku 2017.

Chemické látky môžu byť popísané na rôznej úrovni detailu. InChI zohľadňuje tento fakt a definuje niekoľko vrstiev, kde každá z nich popisuje odlišnú a samostatnú časť štruktúrnej informácie. Vrstvy potom umožňujú postupné spresňovanie popisu danej látky. Momentálne je definovaných nasledujúcich päť vrstiev:

1. **Hlavná vrstva** obsahuje najzákladnejšie informácie o molekule, konkrétne jej chemický vzorec a prepojenie atómov. Ak sa molekula skladá z viacerých grafových komponent, bude mať každá z nich svoj vzorec a vo výslednom zápise budú oddelené bodkou. Nasleduje zápis všetkých väzieb medzi atómami. Väzby s vodíkmi sú vynechané, pokiaľ sa nejedná o vodíkové mostíky. Na záver sú uvedené väzby s imobilnými vodíkmi a normalizovaná poloha prípadných mobilných vodíkov.

2. **Vrstva nábojov** špecifikuje celkový náboj na molekule, čiže rozdiel medzi protónmi a elektrónmi. Ten býva zvyčajne udaný ako skutočný náboj pomocou písmena q, napr. `q+1` pre označenie kladného náboja. U niektorých hydridov sa ale zapisuje ako počet protónov, ktoré treba pridať alebo odobrať z ich neutrálnej formy. V tom prípade sa použije písmeno p, napr. `p-1` pre odobratie protónu, a v prvej vrstve bude uvedený vzorec neutrálneho hydridu.

3. **Stereochemická vrstva** popisuje priestorové usporiadanie molekuly a má dve podvrstvy. Vrstva pre stereochémiu dvojitých väzieb sp^2 sa značí písmenom *b*, zatiaľ čo usporiadanie do štvorstena pri sp^3 sa popisuje vo vrstve *t*.

4. **Vrstva izotopov** spresňuje, ktoré atómy sú izotopy a o aký izotop sa jedná.

5. **Vrstva fixných vodíkov** sa pridáva, ak molekula obsahuje mobilné vodíky, ale užívateľ si možnosť ich presunu nepraje. V takom prípade sa v tejto vrstve dodatočne popíše ich poloha tak, ako je uvedené vo vstupnej štruktúre. Jej význam je hlavne pri rozlišovaní tautomérov a jedná sa o neštandardné rozšírenie.

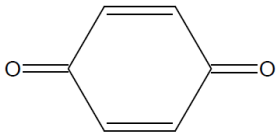
Keďže bude v práci použitá referenčná knižnica implementujúca celý prevod štruktúry až na reťazec, bude proces generovania popísaný len stručne. Úplný popis je možné nájsť v technickom manuáli InChI [37]. Hlavná výhoda využitia referenčnej knižnice spočíva v jej pozícii hlavného rozhodcu v prípadoch, kde môžu nastať nezhody. Ak by sa teda InChI v ojedinelom prípade líšilo medzi implementáciami, za správny výsledok sa bude považovať ten, ktorý vytvorila referenčná knižnica.

Identifikátor je vytváraný zo vstupnej štruktúry v troch krokoch, ktoré znázorňuje obrázok 3.3 a popisujú nasledujúce body:

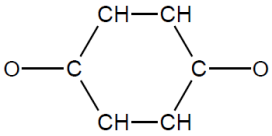
1. **Normalizácia** odstraňuje informácie, ktoré nie sú potrebné pre popis alebo odlíšenie vrstiev. Základným krokom normalizácie je odstránenie typov väzieb a nábojov, ktoré sú ďalej ekvivalentne reprezentované pomocou atómov vodíka. To umožňuje presunúť všetky špecifické vlastnosti na atómy a väzby už len definujú prepojenie atómov. V závislosti na danej látke sú prípadne vykonávané ďalšie kroky normalizácie, ako napríklad odpojenie solí a kovov, eliminácia radikálov alebo detekcia rezonančných foriem.

2. **Kanonizácia** vytvára popis pre atómy danej látky nezávisle na tom, ako bola štruktúra nakreslená. Tiež je zodpovedná za rozdelenie informácie do vrstiev. Generovanie prebieha v krokoch, kedy sa postupne vytvárajú jednotlivé vrstvy. Pri vytváraní každej vrstvy je snaha minimalizovať ju bez toho, aby sa zmenila niektorá z predchádzajúcich.

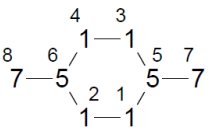
3. **Serializácia** prevádza popis získaný z kanonizácie na textový reťazec.



(a) Vstupná štruktúra



(b) Normalizovaná štruktúra



(c) Kanonické číslovanie

InChI=1S/C6H4O2/c7-5-1-2-6(8)4-3-5/h1-4H

(d) Serializovaný výstup

Obr. 3.3: Proces vytvárania identifikátora InChI [37]

Činnosť knižnice v jednotlivých krokoch je možné riadiť parametrami. Aby sa dosiahlo skutočnej univerzálnosti a interoperability aj medzi rôznymi databázami, boli v roku 2009 zvolené východzie parametre, pri ktorých knižnica generuje štandardný InChI. Ak by si

to však daná aplikácia vyžadovala, parametre je stále možné zmeniť a výstupom bude neštandardný identifikátor.

InChIKey

Kvôli premenlivej a často značnej dĺžke InChI identifikátora bol vytvorený skrátený identifikátor InChIKey, ktorý vznikne z pôvodného InChI použitím hašovacej funkcie [16]. InChIKey má pevnú dĺžku a pozostáva len z veľkých písmen a pomlčiek. InChIKey sa snaží zachovať vrstvy definované v InChI, preto má tri bloky oddelené pomlčkami. Prvý blok, zložený zo 14tich znakov, kóduje prvú vrstvu, teda základný popis molekuly. Ďalší blok, dlhý desať znakov, kóduje všetky ostatné vrstvy okrem p, ktorá je uvedená samostatne. Dva posledné znaky v tomto bloku sú vyhradené pre verziu InChI a označenie, či je InChI štandardné. Na kódovanie vrstiev teda zostáva iba osem znakov. Posledný blok obsahuje len jeden znak, ktorý určuje náboj molekuly, čiže spomínanú vrstvu p. Pre hašovanie sa používa skrátený výstup funkcie SHA-256, zakódovaný do Base26 reprezentácie, teda veľké písmená A–Z. Posledný blok sa nehašuje, ale prevod náboja na písmeno je definovaný tabuľkou. Viac podrobností ohľadom prevodu InChI na InChIKey je možné opäť nájsť v technickom manuály [37]. Ako príklad môže poslúžiť molekula adrenalínu uvedená v predchádzajúcej časti na obrázku 3.2, ktorá má InChIKey `UCTWMZQNUQWSLP-VIFPVBQESA-N`.

Výhodou skrátenej verzie je menšia pamäťová náročnosť, čo umožňuje aj rýchlejšie porovnanie reťazcov, ale hlavne je takýto identifikátor možné použiť naprieč internetom, keďže kódovanie bolo zvolené tak, aby podporovalo populárne vyhľadávače, použitie v URL alebo kopírovanie z dokumentov. Keďže sa však jedná o hašovanú formu InChI, môže dôjsť ku konfliktom, kedy dve rôzne molekuly budú zdieľať rovnaký InChIKey. Pravdepodobnosť konfliktu je v každom prípade veľmi nízka. Minimálna veľkosť databázy, v ktorej sa očakáva konflikt, je približne 2.2×10^{15} záznamov. Táto pravdepodobnosť bola experimentálne overená na databázach obsahujúcich desiatky až stovky miliónov molekúl a tiež na počítačom generovaných sadách v článku [26]. V reálnych databázach konflikty zatiaľ pozorované neboli. Očakáva sa, že k prípadným konfliktom bude dochádzať hlavne v druhom bloku InChIKey pri veľkom počte izomérov, ktoré by mohli vzniknúť pri počítačom generovaných látkach.

3.3 Molekulárne odtlačky

Molekulárne odtlačky sú koncept využívaný v chemoinformatických aplikáciách, ktorý sa snaží riešiť problém náročného grafového porovnávania molekúl. Vo väčšine prípadov sa jedná o bitové polia s pevnou alebo premenlivou dĺžkou, ktoré kódujú prítomnosť alebo absenciu určitých podštruktúr v molekule. Konkrétny spôsob kódovania závisí od druhu molekulárneho odtlačku.

Odtlačky sú postavené na dvoch základných myšlienkach. Prvou je, že porovnanie vhodne zakódovaných bitových polí je výpočtovo menej náročné, než porovnanie grafových reprezentácií molekúl. Tou druhou je, že ak sa predpokladá opakované porovnávanie tak, ako je tomu v prípade databáz, je vhodné si určité informácie o molekule predpočítať.

Používajú sa predovšetkým pre rýchly odhad podobnosti molekúl, ktorý je založený na princípe, že molekuly s podobnou štruktúrou majú podobné chemické vlastnosti. Rozsiahle porovnanie odtlačkov vzhľadom na ich schopnosť klasifikovať molekuly ako štruktúrne podobné bolo vykonané v článku [23]. Na základe tohto porovnania bolo zvolených niekoľko zástupcov molekulárnych odtlačkov, ktoré budú v tejto časti práce predstavené.

3.3.1 Atómové páry

Metóda atómových párov, predstavená v článku [4], bola jedna z prvých popísaných metód pre získanie odtlačkov. Jej princípom je kódovanie vlastností všetkých dvojíc atómov a najkratšej cesty medzi nimi. Atómový pár je definovaný ako podštruktúra zložená z dvoch nevodíkových atómov a väzieb, ktoré ich oddeľujú. Pri atómoch sa uvažujú tri základné vlastnosti, ktorými sú prvok (P), počet väzobných π elektrónov (E) a počet susedných nevodíkových atómov (A). V článku bolo uvažovaných iba trinásť základných prvkov `C`, `O`, `N`, `S`, `F`, `Cl`, `Br`, `I`, `P`, `Si`, `B`, `Se` a `As`. Hodnota P predstavovala index do tabuľky týchto prvkov. Tieto tri vlastnosti sú pre každý atóm k zakódované do jedného 10-bitového celého čísla pomocou vzťahu 3.1.

$$\mathrm{ap}(k) = 64P + 16E + A \tag{3.1}$$

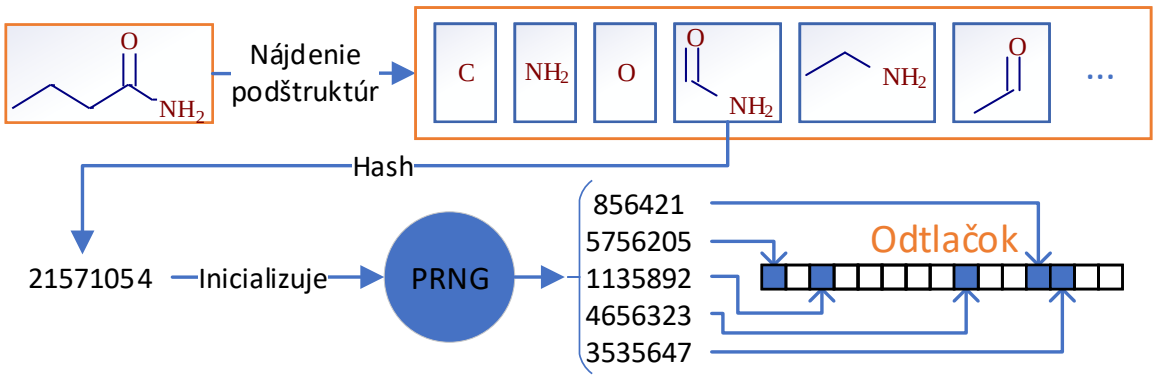
$$\text{atom-pair key} = \min(\mathrm{ap}(j), \mathrm{ap}(k)) + 1024\,(\max(\mathrm{ap}(j), \mathrm{ap}(k)) + 1024D(k)) \tag{3.2}$$

Článok vznikal v čase, kedy bitové polia ešte neboli zaužívaným spôsobom reprezentácie molekulárnych odtlačkov. V článku sa namiesto toho využíva reprezentácia každého páru jedným 32-bitovým celým číslom, vypočítaným pomocou vzťahu 3.2. Hodnoty $\mathrm{ap}(j)$ a $\mathrm{ap}(k)$ zodpovedajú zakódovaným vlastnostiam atómov j a k, a $D(k)$ je dĺžka najkratšej cesty medzi týmito atómami. Odtlačok je tak reprezentovaný ako množina čísel. Prevod na bitový vektor je možný rôznymi spôsobmi. Jedným z nich je uvažovať výslednú množinu čísel ako indexy bitov v bitovom poli, ktorých hodnota sa zmení na 1. Ak sa jedná o bitový vektor s pevnou dĺžkou, rozsah hodnôt je možné obmedziť pomocou operácie modulo.

3.3.2 Odtlačky typu Daylight

Odtlačky tohto typu sú založené na princípe kódovania niekoľkých vzorov, predovšetkým ciest rôznej dĺžky. Celkovo je odtlačok zložený z nasledujúcich podštruktúr [8]:

1. Samostatné atómy.
2. Vzory tvorené atómom a jeho najbližšími susedmi, spolu s väzbami.
3. Atómy a väzby medzi nimi tvoriace cestu v grafe, ktorej dĺžka je menšia ako určitá konštanta.



Obr. 3.4: Proces vytvárania odtlačku Daylight

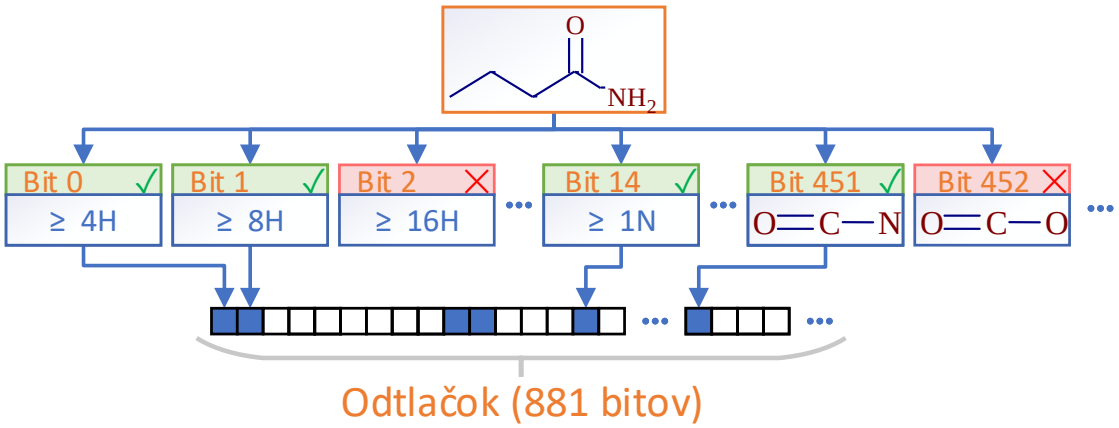
Ako prvé sa vytvorí odtlačok, čiže bitové pole požadovanej dĺžky a všetky bity sa inicializujú na hodnotu 0. Následne sa nájde výskyt všetkých uvedených podštruktúr. Každý

výskyt slúži po zahašovaní k inicializácii pseudo-náhodného generátora náhodných čísel. Následne sa pomocou generátora vygeneruje určený počet čísel, bežne štyri až päť, ktoré po operácii modulo s dĺžkou odtlačku slúžia ako indexy do bitového poľa. Do bitov na týchto pozíciách sa zapíše hodnota 1. Každý výskyt vzoru je teda zakódovaný do niekoľkých bitov. Tento proces pre jednu podštruktúru zachytáva obrázok 3.4.

Spôsob, akým sa výskyt vzoru hašuje, závisí už od konkrétnej implementácie. Tá v prípade pôvodného odtlačku, vyvinutého spoločnosťou Daylight, nie je verejne dostupná, pretože je súčasťou komerčného softvéru. Bola preto snaha vytvoriť odtlačok tohto typu s verejne dostupnou implementáciou, na základe čoho vznikol napríklad odtlačok FP2 v knižnici OpenBabel alebo RDKit Fingerprint v knižnici RDKit.

3.3.3 CACTVS

CACTVS je molekulárny odtlačok, ktorý spadá do kategórie tzv. štruktúrnych kľúčov (angl. structural keys). U predchádzajúcich odtlačkov nemali bity žiadny konkrétny význam a mohla nastať situácia, že jeden bit bol nastavený viackrát. V prípade štruktúrnych kľúčov reprezentuje každý bit určitú vlastnosť alebo výskyt preddefinovaného vzoru. Ak je hodnota bitu 1, molekula túto vlastnosť spĺňa, respektíve sa v nej vzor nachádza. Rôzne odtlačky z tejto kategórie sa potom líšia hlavne významom jednotlivých bitov.



Obr. 3.5: Princíp vytvárania CACTVS odtlačku na niekoľkých vybraných bitoch. Znázornené bity 451 a 452 sú pomocou jazyka SMARTS popísané ako `C(-N)(=O)` a `C(-O)(=O)`.

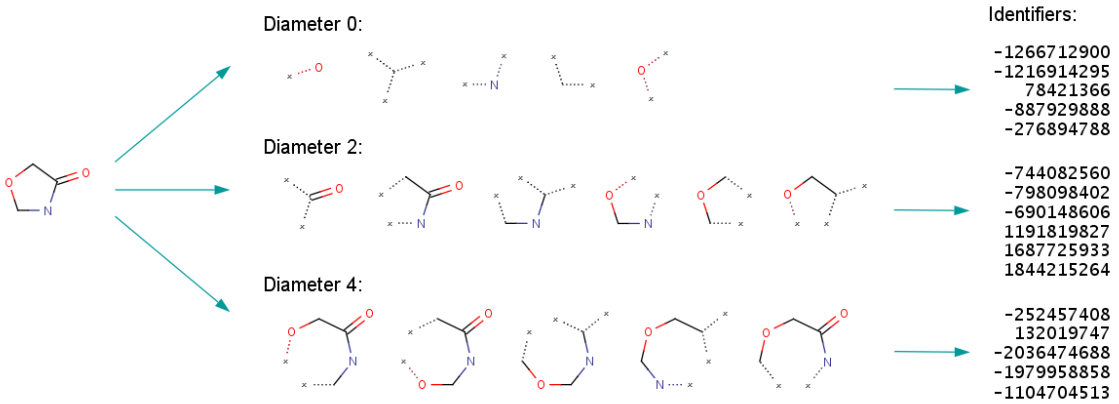
V prípade CACTVS je možné nájsť význam všetkých 881 bitov v online dokumente [29]. Prvých 263 bitov predstavuje rôzne dopyty na vlastnosti molekuly, napríklad či je celkový počet vodíkov aspoň štyri (bit 0), alebo či molekula obsahuje aspoň jeden cyklus veľkosti tri (bit 115). Zvyšné bity značia, či sa v molekule nachádzajú konkrétne vzory, ktoré sú popísané pomocou jazyka SMARTS predstaveného v sekcii 3.2.1. Obrázok 3.5 znázorňuje princíp získania odtlačku pre jednu štruktúru.

3.3.4 Kruhové odtlačky ECFP

Kruhové odtlačky sú kategória odtlačkov popisujúce molekulu ako podštruktúry, ktoré vzniknú uvažovaním atómu a jeho susedov v kruhovom okolí, ktoré sa postupne rozširuje. Extended Connectivity Structural Fingerprint (ECFP) je zástupcom tohto typu odtlačkov popísaný v článku [34], na ktorom je možné jednoducho vysvetliť princípy tejto metódy. Proces vytvárania ECFP má tri kroky:

1. Prvotné priradenie celočíselného identifikátora každému atómu.
2. Iteratívna aktualizácia identifikátorov na základe zohľadnenia okolitých susedov.
3. Odstránenie duplikátov.

V prvom kroku je každému atómu priradený identifikátor, ktorý vznikne zahašovaním šiestich vlastností do jedného 32-bitového čísla. Týmito vlastnosťami sú počet priamo susediacich atómov, ktoré sú ťažkými prvkami, počet naviazaných vodíkov, valencia bez počtu vodíkov, protónové číslo, atómová hmotnosť a či je atóm súčasťou aspoň jedného cyklu.



Obr. 3.6: Proces vytvárania ECFP identifikátorov[1]

V druhom kroku dochádza k aktualizácii identifikátorov nasledujúcim spôsobom. Pre každý atóm sa vytvorí pole, do ktorého je vložené číslo iterácie a identifikátor tohto atómu. Následne sa do poľa postupne pridávajú identifikátory atómov susediacich s aktuálnou podštruktúrou a typ väzby, ktorá ich spája. V prvej iterácii je aktuálna podštruktúra iba počiatočný atóm, ale v druhej iterácii sú to už aj atómy, ktorých vzdialenosť od počiatočného atómu je jedna, a v ďalších iteráciách sa okolie stále viac rozrastá. Po vložení všetkých susediacich atómov sa pole opäť zahašuje do jedného 32-bitového čísla, ktoré sa stáva novým identifikátorom daného atómu. Nové identifikátory sa priradia naraz všetkým atómom až na konci iterácie. Tento princíp zjednodušene ilustruje obrázok 3.6. V článku sa neuvádza konkrétna hašovacia funkcia, ktorá má byť použitá. Dôležité je iba to, aby priraďovala poliam celých čísel identifikátory náhodne a rovnomerne.

Po dosiahnutí požadovaného počtu iterácií sa identifikátory zo všetkých iterácií zozbierajú a odstránia sa duplikáty. Odtlačok je potom možné reprezentovať ako množinu výsledných identifikátorov alebo bitové pole, kde každý identifikátor predstavuje index bitu, ktorého hodnota má byť zmenená na 1.

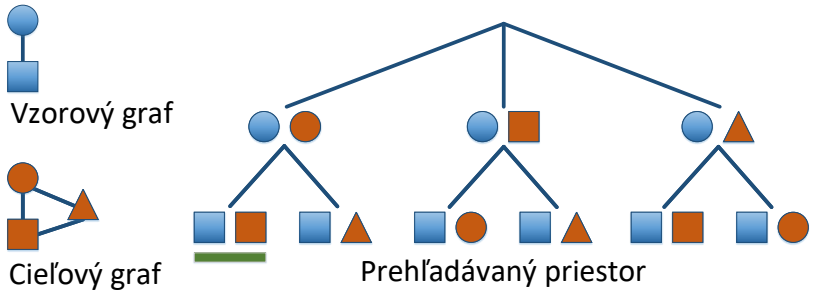
3.4 Algoritmy pre podštruktúrne vyhľadávanie

Úlohou podštruktúrneho vyhľadávania je nájsť požadovaný vzor v chemickej štruktúre. Táto úloha zodpovedá v teórii grafov nájdeniu izomorfného podgrafu. Neindukovaný podgrafový izomorfizmus je injektívne zobrazenie $i : G_\alpha \to G_\beta$ grafu $G_\alpha = (V_\alpha, E_\alpha, \varepsilon_\alpha)$ do grafu

[1]Zdroj obrázka: https://docs.chemaxon.com/display/docs/Extended+Connectivity+Fingerprint+ECFP

$G_\beta = (V_\beta, E_\beta, \varepsilon_\beta)$, ktoré zachováva susednosť vrcholov, teda ak existuje hrana medzi vrcholmi $u, v \in V_\alpha$, musí existovať hrana aj medzi vrcholmi $i(u), i(v) \in V_\beta$. Pre indukovaný izomorfizmus navyše platí, že ak hrana medzi u, v neexistuje, nesmie existovať ani medzi $i(u), i(v)$. Pokiaľ nebude explicitne uvedené inak, tak sa vo zvyšku práce pod hľadaním izomorfného podgrafu myslí práve hľadanie neindukovaného podgrafového izomorfizmu. V nasledujúcom texte bude G_α nazývaný vzorový graf a G_β cieľový graf.

Z pohľadu teórie grafov musí platiť pre každý vrchol podmienka, že stupeň vrcholu v G_α musí byť menší alebo rovný stupňu zodpovedajúceho vrcholu v G_β. Niektoré algoritmy toto využívajú pre rýchlejšie prehľadávanie priestoru. Z hľadiska chémie je ale navyše väčšinou žiadané, aby sa zhodovali aj všetky chemické vlastnosti popísané v predošlých častiach, napríklad prvok či izotop. Ďalej je nutné uvažovať, že hrany predstavujúce väzby v chemických štruktúrach už nie sú zameniteľné tak, ako je tomu u obyčajných grafov.



Obr. 3.7: Prehľadávaný priestor ako strom. Zelené podtrhnutie označuje vetvu, v ktorej dôjde k nájdeniu izomorfizmu. [2]

Problém izomorfizmu podgrafu je možné riešiť priamočiarym spôsobom, kedy sa postupne prechádzajú všetky možnosti priradenia vrcholov vzoru na cieľový graf. Priradenie jedného vrcholu a následné rekurzívne priraďovanie jeho susedov potom zodpovedá prechodu stromom, ktorý zobrazuje obrázok 3.7.

3.4.1 Ullmannov algoritmus

Ullmann bol prvým, kto sa snažil toto vyhľadávanie zefektívniť [39]. Jeho algoritmus, popísaný pseudokódom v algoritme 1, je postavený na prechode stromom do hĺbky, pričom ho postupne prerezáva na základe splnenia podmienky 3.3. K vysvetleniu pseudokódu je nutné zaviesť ešte niekoľko ďalších notácií. Nech $a_{ix} = 1$ ak vrcholy v_i a v_x spolu susedia v G_α a obdobne $b_{jy} = 1$ znamená susednosť v G_β. Ďalej M je matica možných priradení a m_{ij} jej prvok, kde i je index vrcholu vzoru a j index vrcholu v cieľovom grafe. Podmienka potom hovorí, že mapovanie i-tého vrcholu vzoru $v_{\alpha i} \in V_\alpha$ k j-tému vrcholu cieľového grafu $v_{\beta j} \in V_\beta$ je možné iba vtedy, ak pre akýkoľvek vrchol zo vzorového grafu $v_{\alpha x}$ susediaci s $v_{\alpha i}$ existuje taký vrchol $v_{\beta y}$ susediaci s $v_{\beta j}$, že $v_{\alpha x}$ je možné priradiť k $v_{\beta y}$.

$$\underset{0 \leq x < |V_\alpha|}{\forall} x : (a_{ix} = 1 \implies \underset{0 \leq y < |V_\beta|}{\exists} y : (m_{xy} \cdot b_{jy} = 1)) \tag{3.3}$$

V prípade, že podmienka nie je splnená, algoritmus už nemusí skúšať priradenie zvyšných vrcholov, čo znamená, že sa vynechá celý podstrom s koreňom v aktuálnom priradení. Čím skôr sa toto udeje, tým väčšiu časť priestoru nebude potrebné prehľadávať. Tento prístup zodpovedá metóde spätného navrátenia (angl. backtracking), používanej pri riešení problémov s obmedzujúcimi podmienkami.

Algoritmus 1 Ullmannov algoritmus

```
1:  procedure CONDITION(i, j, M)
2:      for x := 0 to |V_α| do
3:          if a_ix = 1 then
4:              for y := 0 to |V_β| do
5:                  if m_xy = 1 and b_jy = 1 then
6:                      break
7:                  return false
8:      return true
9:
10: procedure REFINE(M)
11:     repeat
12:         for all m_ij ∈ M do
13:             if m_ij = 1 and not CONDITION(i, j, M) then
14:                 m_ij ← 0
15:     until M bez zmeny
16:     if M nebolo zmenené then
17:         return true
18:     else
19:         return false
20:
21: procedure SEARCH(M, d, F)
22:     if ∄j : m_dj = 1 and F_j = 0 then
23:         return false
24:     M' ← M
25:     for all k < |V_β| do
26:         if m_dk = 0 or F_k = 1 then
27:             continue
28:         for all j ≠ k do
29:             m_dj ← 0
30:         if REFINE(M) then
31:             if d = |V_α| or SEARCH(M, d + 1, F) then
32:                 return true
33:         M ← M'
34:         if ∄j : j > k and m_dj = 1 and F_j = 0 then
35:             return false
```

V pôvodnom texte je algoritmus uvedený vo forme, ktorá používa niekoľko stavov a mnoho `goto` príkazov, pomocou ktorých medzi týmito stavmi prechádza. Z pohľadu súčasného spôsobu popisu toku programu je však táto forma zápisu nevhodná. Algoritmus bol preto prepísaný na niekoľko procedúr a cyklov s bežnou notáciou. Ďalším dôvodom pre uvedenie plného algoritmu je, že mnohé ďalšie práce z neho vychádzajú, pričom myšlienka priraďovania vrcholov zostáva zachovaná a mení sa len spôsob, akým je redukovaný prehľadávaný priestor. Z uvedeného pseudokódu je možné vidieť, že sa jedná o rekurzívny algoritmus. Za účelom zjednodušenia zápisu nie sú algoritmu samotné grafy G_α a G_β predávané ako parameter, ale sú globálne dostupné. Algoritmus začína volaním procedúry SEARCH a parametrami M^0, $d^0 = 0$ a F^0. F^0 je nulový vektor o veľkosti $|V_\beta|$ a M^0 je

matica veľkosti $|V_\alpha| \times |V_\beta|$, pre ktorú platí:

$$m_{ij}^0 = \begin{cases} 1 & \text{ak stupeň vrcholu } v_{\alpha i} \text{ je menší ako stupeň vrcholu } v_{\beta j} \\ 0 & \text{inak} \end{cases}$$

Ak existuje priradenie vrcholov z G_α do G_β, čiže ak je G_α podgraf, procedúra skončí úspešne a matica M bude obsahovať hodnotu 1 v každom riadku práve jedenkrát. V takom prípade bude prvok $m_{ij} = 1$ značiť, že i-tý vrchol z V_α zodpovedá j-tému prvku z V_β.

Ullmann tento algoritmus popísal už v roku 1976 [39]. Jeho hlavným problémom je veľká pamäťová náročnosť, až $\Theta(N^3)$ v najlepšom aj najhoršom prípade [6]. Ďalšie algoritmy tento problém úspešne riešili a Ullmann tiež v roku 2010 publikoval rozsiahlu prácu [40], v ktorej okrem iného nadväzuje na svoj predchádzajúci algoritmus a zlepšuje jeho pamäťovú, ale aj časovú náročnosť. K tomu využíva optimalizované dátové štruktúry, paralelizmus na úrovni bitových operácií a tiež výsledky ostatných výskumných skupín, ktoré sa problému hľadania izomorfného podgrafu a príbuzným oblastiam medzičasom venovali, napríklad Boussemartovu heuristiku pre dynamické radenie premenných [3].

3.4.2 VF2

VF je algoritmus pre hľadanie izomorfných grafov a podgrafov vytvorený v snahe prísť s lepším algoritmom ako Ullmannov, ktorý bol v čase vytvorenia VF stále jedným z najpoužívanejších vďaka jeho všeobecnosti a efektivite. VF bol popísaný v článku [5]. V tejto práci bude predstavená jeho vylepšená verzia, VF2, ktorá oproti pôvodnému algoritmu výrazne optimalizuje pamäťové nároky. Pseudokód v algoritme 2 popisujúci túto novšiu verziu vychádza z článku [6].

Algoritmus 2 VF2

```
1:  procedure MATCH(s)
2:      if M(s) pokrýva všetky vrcholy V_α then
3:          return M(s)
4:      else
5:          P(s) ← vypočítaná množina možných párov pre zahrnutie do M(s)
6:          for all (n, m) ∈ P(s) do
7:              if F(s, n, m) then
8:                  s' ← stav získaný pridaním (n, m) do M(s)
9:                  MATCH(s')
10:         Obnov dátové štruktúry do predchádzajúceho stavu
```

Algoritmus začína predaním počiatočného stavu s_0, pre ktorý platí $M(s_0) = \emptyset$. Výstupom algoritmu je priradenie vrcholov medzi grafmi ako množina dvojíc M. Podobne ako Ullmannov algoritmus, VF2 rekurzívne prehľadáva priestor do hĺbky a pri nájdení stavu nevedúceho k riešeniu využíva spätného návratu. Pre popis a riešenie problému je však použitá tzv. stavovo-priestorová reprezentácia (angl. state space representation, SSR), v ktorej je každý stav asociovaný s čiastočným riešením priradenia vrcholov $M(s)$, ktoré je podmnožinou celkového riešenia M. K jej implementácii postačuje šesť polí, tri o veľkosti $|V_\alpha|$ a tri o veľkosti $|V_\beta|$. Počas práce algoritmu nedochádza k ich duplikovaniu, vďaka čomu je pamäťová náročnosť iba $O(N)$ vzhľadom na počet vrcholov v grafoch. Dôležitým rozdielom je tiež absencia inicializačnej fázy, vďaka čomu dokáže vyriešiť triviálne prípady oveľa rýchlejšie než Ullmannov algoritmus.

K prerezávaniu stromu dochádza aplikovaním piatich obmedzujúcich podmienok, ktoré je možné nájsť popísané v pôvodnom článku. Prvé dve overujú správnosť aktuálneho čiastočného riešenia a ďalšie tri vykonávajú dopredné overenie (angl. look-ahead). Ich logickým prienikom vzniká funkcia $F_{syn}(s, n, m)$. Ďalej je definovaná funkcia F_{sem}, ktorá má význam, ak sú vrcholom alebo hranám priradené atribúty. V takom prípade musia byť hodnoty atribútov v priradení zhodné. Funkcia $F(s, n, m) = F_{syn} \wedge F_{sem}$ potom určuje, či aktuálny stav vedie k riešeniu, teda či sa má z neho pokračovať ďalej.

Implementáciu algoritmu VF2 je možné nájsť ako súčasť knižnice VFLib[1]. V roku 2012 bolo vykonané porovnanie Ullmannovho algoritmu a VF2 založené na vyhľadávaní SMARTS vzorov v databáze ZINC obsahujúcej mnoho chemických látok [12]. Meraním bolo zistené, že VF2 dokáže takmer vždy nájsť daný vzor v grafe rýchlejšie ako Ullmannov algoritmus. Pre prípady, v ktorých trvalo VF2 hľadanie dlhšiu dobu, boli SMARTS vzory preformulované, čím sa dosiahlo aj u týchto rýchlejšieho vyhľadávania oproti Ullmannovmu algoritmu. Autori z tohto dôvodu vytvorili okrem originálnej sady vzorov aj sadu optimalizovaných a naopak anti-optimalizovaných vzorov, pre ktoré taktiež vykonali merania. Zistili, že pri algoritme VF2 môžu vhodne formulované vzory spôsobiť až 15-násobné zrýchlenie a naopak extrémne zle formulované vzory 13-násobné spomalenie. Pri Ullmannovom algoritme boli rozdiely medzi týmito troma sadami zanedbateľné. Výsledkom ich pozorovaní je, že oba algoritmy sú použiteľné pre väčšinu chemoinformatických aplikácií, ale odporúčajú použiť VF2, pretože vo všeobecnosti dosiahne výsledku v kratšom čase. Z článku ale nie je úplne zrejmé, či autori pre porovnanie použili už aktualizovanú verziu Ullmannovho algoritmu. Z citovaných zdrojov by vyplývalo, že sa jednalo ešte o pôvodný algoritmus.

3.4.3 Glasgow

Algoritmus Glasgow bol predstavený v článku [20], ale tento názov bol preň zavedený až v nadväzujúcej práci [18]. Jedná sa o algoritmus, ktorý využíva takzvané doplnkové grafy (angl. supplemental graphs) k vytvoreniu nových obmedzujúcich podmienok. Okrem toho je schopný spätného návratu až o niekoľko stavov, čo býva označované ako spätné skákanie (angl. backjumping). Algoritmus bol navrhnutý tak, aby umožňoval paralelizmus na bitovej úrovni ako aj použitie viacerých vlákien.

Doplnkové grafy je v krátkosti možné predstaviť ako nové grafy, ktoré vznikli z pôvodného uplatnením nasledujúceho pravidla. V doplnkovom grafe $G^{[k,l]}$ existuje medzi dvoma vrcholmi hrana práve vtedy, keď v pôvodnom grafe medzi týmito vrcholmi existuje aspoň k ciest dĺžky l. Vďaka doplnkovým grafom je možné odhaliť vo vyhľadávaní stav, ktorý nevedie k riešeniu, oveľa skôr. Na druhú stranu je ich zostrojenie časovo náročné. Možným riešením pri opakovanom behu algoritmu by bolo ich uloženie na disk. Dvojice pôvodných a doplnkových grafov spárovaných podľa parametrov k a l tvoria množinu grafových párov L.

Pri popise algoritmu predstavuje doména D_v množinu možných priradení vrcholu vzorového grafu $v \in V_\alpha$ na vrcholy cieľového grafu V_β. Toto označenie pochádza z teórie programovania s obmedzujúcimi podmienkami. Pre každý vrchol z V_α existuje jedna doména, ktoré spolu tvoria množinu D. Ďalej sa v algoritme vyskytuje množina konfliktných vrcholov F, pre ktoré neexistuje riešenie vzhľadom na už existujúce priradenia. Táto množina sa používa k určeniu bodu riešenia, do ktorého je potrebné sa vrátiť, aby sa konflikt odstránil.

Hlavnú časť algoritmu popisuje pseudokód v algoritme 3, ktorý je až na zmenu značenia prebratý z článku [20]. Jedná sa opäť o rekurzívny algoritmus, ktorý skončí úspechom, ak je

[1] Odkaz na stiahnutie: http://www3.cs.stonybrook.edu/~algorith/implement/vflib/implement.shtml

Algoritmus 3 Glasgow

```
1:  procedure SEARCH(L, D)
2:      if D = ∅ then
3:          return Success
4:      D_v ← najmenšia doména v D podľa možností priradenia
5:      F ← {v}
6:      for all v' ∈ D_v zoradené podľa stupňa vrcholu v G_β do
7:          D' ← D
8:          case ASSIGN(L, D', v, v') of
9:              Fail F' then F ← F ∪ F'
10:             Success then
11:                 case SEARCH(L, D' \ {D_v}) of
12:                     Success then return Success
13:                     Fail F' then
14:                         if ∄w ∈ F' : D_w ≠ D'_w then
15:                             return Fail F'
16:                         F ← F ∪ F'
17:     return Fail F
```

množina domén D prázdna. V opačnom prípade sa vyberie ďalšia doména podľa zvoleného kritéria a bude sa hľadať priradenie vrcholov v a v', ktoré je zlučiteľné s už vytvorenými priradeniami. Toto priradenie overuje a vykonáva funkcia ASSIGN, ktorá je podrobne popísaná v odkazovanom článku. Jej cieľom je odstrániť vrchol v' z ostatných domén, čím zabraňuje jeho opakovanému priradeniu. Funkcia tiež vynúti v doménach susednosť s práve priradenými vrcholmi. Ak je totiž v susedné s w v G_α, v doméne D_w môžu ostať len možnosti priradenia k takým vrcholom, ktoré sú susedné s aktuálne priradeným vrchol v' v G_β. Ak nastane, že je nejaká doména prázdna, čo by znamenalo, že pre nejaký vrchol už nie je možné žiadne ďalšie priradenie, funkcia skončí neúspechom a dotyčný vrchol bude vrátený ako množina F'.

Ak nie je možné nájsť žiadne priradenie pre vrchol v, funkcia SEARCH končí na riadku 17 neúspechom a vracia svoju množinu F. Dochádza teda k vynoreniu na riadku 13 a nasleduje podmienka, ktorá umožňuje spätné skákanie. Jej zmysel je možné chápať nasledovne: Ak sa doména pre ktorýkoľvek vrchol w z konfliktnej množiny F' v aktuálnom priradení nezmenila, tak potom nie toto, ale už iné priradenie pred ním spôsobilo zlyhanie. V takom prípade nemá zmysel skúšať iné priradenia v tomto zanorení, pretože by skončili taktiež neúspechom. Z tohto dôvodu je možné opäť vrátiť neúspech a prejsť na prechádzajúcu úroveň rekurzie.

V oboch uvedených zdrojoch bolo vykonané porovnanie algoritmu Glasgow s VF2 a ďalšími novšími algoritmami ako sú napríklad SND a LAD. Dátovú sadu v prvom článku [20] tvorili grafy rôznych typov a obsahovala okrem bežných testovacích párov aj zámerne ťažko riešiteľné dvojice a prípady, kedy riešenie neexistuje. Tieto grafy pochádzali jednak z databáz bežne používaných pre porovnania tohto typu algoritmu, napríklad Stanfordská grafová databáza a databáza knižnice VFLib, ktorá bola použitá pri porovnaní VF2 s ostatnými algoritmami, ale pribudli tiež grafy z domény spracovania obrazu. Pre predstavu najväčší vzor obsahoval 900 vrcholov a 12410 hrán. Glasgow bol celkovo najlepším algoritmom pre hľadanie izomorfizmu, ale toto prvenstvo závisí od doby behu. V porovnaní s VF2 bol Glasgow rýchlejší, ak jeho paralelná implementácia riešila problém dlhšie ako 0.06s. Pri sekvenčnej implementácii je táto hranica 0.6s. Doplnkové grafy majú prínos až

pri problémoch, ktorých čas riešenia presahuje 10s. Je to hlavne z dôvodu, že musia byť najskôr zostrojené, čo sa tiež započítavalo do merania. Problémom tohto porovnania je, že dátovú sadu tvorili až príliš náročné problémy. Chemické štruktúry len málokedy dosahujú stovky atómov a počet väzieb, na ktorých sa atóm môže podieľať, je obmedzený. Taktiež doba riešenia jedného problému by jednoznačne nemala presahovať niekoľko miliseкúnd. Z nameraných výsledkov je preto nejasné, nakoľko bude tento algoritmus vhodný pre zamýšľané využitie a bude nutné vykonať vlastné merania.

3.4.4 RI

Algoritmus RI bol predstavený talianskou výskumnou skupinou v článku [2]. Jedná sa o algoritmus pre hľadanie izomorfného podgrafu, ktorý bol vytvorený za účelom využitia predovšetkým v biochemických aplikáciách. Jeho myšlienkou je zostavenie a použitie vyhľadávacej stratégie, ktorá umožní výrazne zmenšiť prehľadávaný priestor bez použitia komplikovaných pravidiel alebo procedúr pre redukciu domén.

Táto stratégia je založená na vhodnom radení premenných problému s obmedzujúcimi podmienkami, čiže vrcholov vzorového grafu. Stratégia je vytvorená čisto na základe topológie vzorového grafu, takže je možné ju opätovne použiť pri viacerých vyhľadávaniach rovnakého podgrafu. Jednotlivé premenné sú zoradené tak, aby sa čo najviac podmienok vyhodnotilo v procese vyhľadávania čo najskôr. Všeobecne je možné princíp radenia použitý v algoritme RI charakterizovať tak, že vrcholy s vysokým počtom hrán a susediace s čo najviac vrcholmi, ktoré už boli zoradené, sa vo výslednom poradí vyskytnú skôr. Samotné radenie potom prebieha tak, že sa vytvorí fronta, do ktorej sa spočiatku vloží vrchol s najväčším počtom susedov. Z tejto fronty sa potom vždy zvolí najlepší vrchol podľa uvedeného princípu radenia, pridá sa do výsledného poradia a jeho susedia, ktorí ešte neboli vložení do fronty, sa tam vložia. Radenie končí vyprázdnením fronty. Okrem poradia sa pritom pre každý vrchol okrem počiatočného určí predok, ktorý ho do fronty pridal.

Algoritmus 4 RI

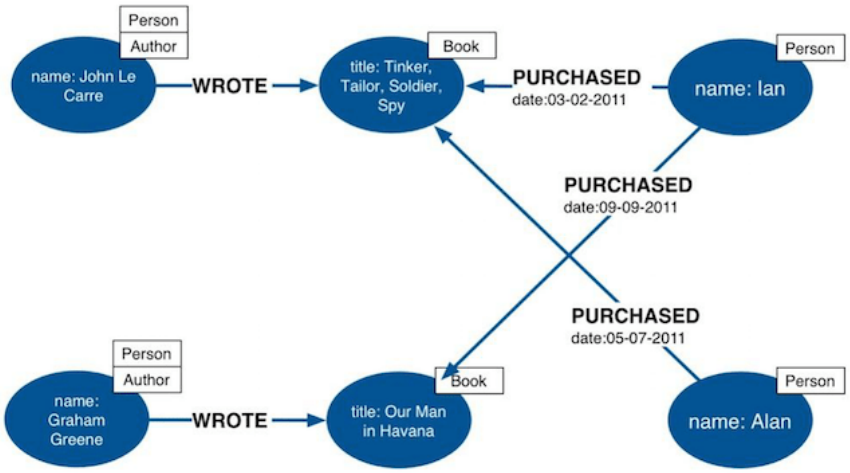
```
1:  procedure SOLVE
2:      s ← 0; s′ ← −1
3:      while s ≥ 0 do
4:          if s′ ≥ s then
5:              v ← R_s; M ← M \ {v}
6:          c ← nasledujúce x ∈ C_s : x ∉ M ∧ COMPATIBLE(x, s)
7:          if ∄c then
8:              s′ ← s; s ← s − 1
9:          else
10:             R_s ← c
11:             if s = |V_α| − 1 then
12:                 Ulož priradenie v R ako výsledok
13:                 s′ ← s
14:             else
15:                 M ← M ∪ {c}
16:                 s′ ← s; s ← s + 1
17:                 t ← predok stavu s v radení premenných
18:                 C_s ← susedia vrcholu R_t                ▷ Vrchol R_t ∈ V_β
```

V pôvodnom článku je pseudokód procedúry pre vyhľadávanie napísaný pomerne abstraktne pomocou iterácií ciest stavového priestoru. Pseudokód v algoritme 4 predstavuje alternatívny zápis, ktorý postupne prechádza stavový priestor a v prípade stavu, ktorý nevedie k riešeniu, využíva spätného návratu (riadok 8). Funkcia SOLVE nájde všetky výskyty hľadaného vzoru. Každému stavu zodpovedá určitý vrchol z V_α a toto priradenie je dané radením, ktoré bolo vykonané pred spustením funkcie SOLVE. Tento zápis ďalej používa niekoľko pomocných premenných. Pole R o veľkosti $|V_\alpha|$ ukladá pre každý stav aktuálne priradený vrchol cieľového grafu a po úspešnom nájdení izomorfizmu je možné z neho získať výsledné priradenie vrcholov z G_α do G_β. Množina M obsahuje tie vrcholy patriace do V_β, ktoré už sú súčasťou aktuálneho priradenia a nie je ich teda možné znova priradiť. Pole kandidátnych množín C obsahuje pre každý stav s množinu kandidátnych vrcholov patriacich do V_β, ktoré má zmysel priradiť k vrcholu z V_α zodpovedajúcemu tomuto stavu. Funkcia COMPATIBLE na riadku 6 kontroluje, či je daný kandidát kompatibilný so súčasným stavom, teda či sa zhodujú vlastnosti vrcholov a ich hrán.

3.5 Grafové databázy

Databáza je systém pre ukladanie a spätné získavanie dát. Databázy je podľa konceptu možné rozdeliť na dve veľké skupiny, relačné a NoSQL. V relačných databázach sú dáta uložené v tabuľkách definovaných reláciami. NoSQL databázy svoje dáta ukladajú odlišným spôsobom, napríklad ako páry kľúč-hodnota, kde všetky záznamy sú prvkom vo veľkom asociatívnom poli. Dôležitým aspektom NoSQL databáz je často absencia schémy. Nové informácie je tak možné pridávať dynamicky podľa potreby a jednotlivé záznamy nemusia obsahovať rovnaké polia tak, ako je to v relačných databázach [33].

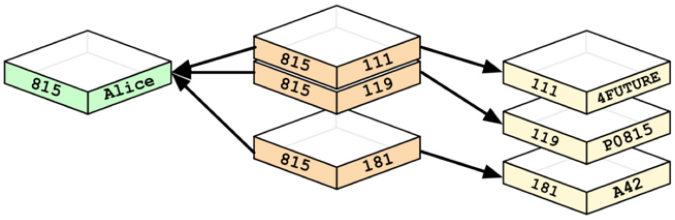
Grafové databázy sú NoSQL databázové systémy podporujúce základné operácie vytvorenia, čítania, aktualizácie a mazania záznamov (CRUD), ktoré pracujú nad grafovo-orientovaným dátovým modelom. Databázy tohto typu vychádzajú z teórie grafov, ktorú využívajú pri ukladaní, získavaní a spracovávaní dát. Niektoré grafové databázy ukladajú dáta prirodzene ako grafy v optimalizovanom úložisku. To však nie je podmienkou a u iných databáz sa grafy prevádzajú napríklad do objektovo-orientovaného úložiska [33].



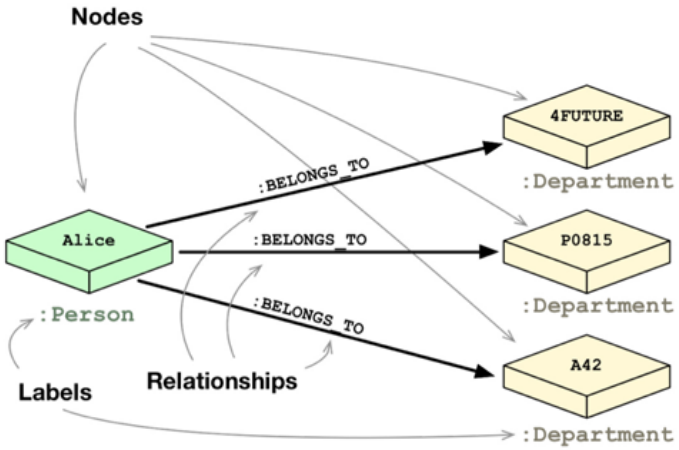
Obr. 3.8: Príklad atribútového grafu[1]

[1] https://s3.amazonaws.com/dev.assets.neo4j.com/wp-content/uploads/property_graph_model.png

Jednou z najpopulárnejších grafových databáz je Neo4j [36], ktorá bude slúžiť ako referenčná pri posudzovaní využiteľnosti tohto konceptu. Neo4j definuje a implementuje model takzvaného atribútového grafu (angl. Labeled Property Graph), ktorý je rozšírením matematickej definície grafu. Model umožňuje vrcholom pridať ľubovoľný počet návestí, pomocou ktorých ich je možné kategorizovať. Hrany musia mať orientáciu a práve jedno návestie, ktoré určuje typ vzťahu. Ďalej je možné vrcholom aj hranám pridať jeden alebo viac atribútov a následne priradiť týmto atribútom hodnotu. Dáta sú teda uložené vo forme orientovaných grafov, v ktorých vrcholy aj hrany nesú určitú informáciu [38]. Vrcholy zvyknú reprezentovať objekty, ako sú napríklad osoby alebo veci. Hrany potom vyjadrujú vzťahy medzi nimi, napríklad známosť alebo vlastníctvo. Jednoduchý prípad práve popísaného grafu je možné vidieť na obrázku 3.8.



(a) Relačná databáza



(b) Grafová databáza

Obr. 3.9: Rozdiel v modelovaní vzťahov medzi relačnou a grafovou databázou [38]

Grafové databázy vynikajú v schopnosti pracovať s dátami s vysokou mierou prepojenosti. Je to vďaka tomu, že vzťahy medzi entitami sú pre ne nemenej dôležité než informácie o entitách samotných. Grafová štruktúra je pre tento prípad veľmi vhodnou reprezentáciou, ktorá umožňuje efektívny prechod vrcholmi na základe vzťahov [38]. Toto je jednou z výhod grafových databáz oproti relačným, v ktorých by prechod zodpovedal netriviálnej operácii zlúčenia záznamov rôznych tabuliek na základe kľúčov [33]. Rozdiel v modelovaní vzťahov medzi relačnými a grafovými databázami najlepšie vyjadrujú obrázky 3.9a a 3.9b.

```
1 MATCH (n:Person)-[r:PURCHASED]->(m)
2 WHERE n.name='Ian'
3 RETURN n,r,m
```

Ukážka 3.1: Príklad dopytu v jazyku Cypher

Dopytovanie konkrétne v Neo4j prebieha pomocou špeciálneho jazyka nazývaného Cypher. Jedná sa o deklaratívny, grafovo-orientovaný dopytovací jazyk s prvkami SQL. Svojou syntaxou pripomína ASCII kresbu vzťahu, ktorý vytvára alebo sa snaží z databázy získať [38]. Napríklad dopyt v ukážke 3.1 zodpovedá dvom vrcholom, z ktorých jeden spadá do kategórie osoba a navyše jeho atribút meno musí mať hodnotu Ian. Medzi vrcholmi potom musí byť vzťah *zakúpený*. Takémuto predpisu vyhovujú v obrázku 3.8 dve možnosti, kde v oboch prípadoch si Ian zakúpil knihu.

Kapitola 4

Analýza požiadaviek na systém

Táto kapitola je venovaná definícii požiadaviek, ktoré by mal systém predstavený v úvode práce spĺňať. K tomu je nutné najskôr bližšie objasniť motiváciu, z ktorej tieto požiadavky vyplývajú. Tomu sa venuje časť 4.1. V rámci definície požiadaviek je taktiež vhodné predstaviť niektoré z existujúcich riešení, ktoré môžu slúžiť v určitom ohľade ako inšpirácia alebo základ pri vyhodnocovaní efektivity výsledného riešenia. V časti 4.2 je preto predstavený jeden databázový systém podporujúci vyhľadávanie chemických štruktúr a jedna vybraná chemoinformatická knižnica, ktorá implementuje viacero algoritmov potrebných pri vytváraní takéhoto systému. Následne je časť 4.3 venovaná už samotnej definícii požiadaviek, ktoré majú byť splnené.

4.1 Motivácia

Základným cieľom tejto práce je vytvorenie systému, ktorý umožní rýchlo a efektívne vyhľadávať štruktúry vo veľkých databázach chemických látok. Vyhľadávanie je jednou zo základných operácií, vďaka ktorej je možné získať potrebné dáta z databáz a ďalej s nimi pracovať. Túto úlohu komplikujú hlavne dva aspekty. Prvým je veľký a stále narastajúci[1] počet záznamov v chemických databázach. Rádovo sa jedná o desiatky miliónov rôznych látok. Tým druhým je špecifická forma dát. Bežne sa v databázach vyhľadáva na základe číselných alebo textových hodnôt, pri ktorých je operácia porovnania triviálna alebo algoritmicky nenáročná. Ako však bolo uvedené v prechádzajúcich kapitolách, najbežnejším spôsobom popisu chemických látok je ich štruktúrny vzorec, pre ktorý existuje v informačných systémoch prirodzená reprezentácia formou grafu. A práve porovnanie dvoch grafov zmysluplným spôsobom, napríklad na zhodu, podobnosť alebo výskyt podgrafu, je náročná operácia. Aj v prípade nepochybne veľmi rýchleho algoritmu, ktorý by bol schopný porovnať dva grafy rádovo v mikrosekundách, by porovnanie všetkých záznamov vo veľkej databáze trvalo desiatky sekúnd. Keďže sa jedná o čas, ktorý sa z pohľadu užívateľa blíži hranici únosnosti, bude nutné nájsť spôsoby, ktoré umožnia tieto problémy riešiť efektívnejšie.

Väčšina veľkých databáz dostupných online, napríklad databáza PubChem popísaná bližšie v časti 4.2.1, štruktúrne vyhľadávanie v nejakej forme podporuje. Navyše majú často dostupné aj aplikačné rozhranie (API), ktoré umožňuje ich použitie v samostatných aplikáciách. Naskytá sa preto otázka, aký zmysel má vytvárať vlastné riešenie.

[1] Oficiálne štatistiky o vývoji veľkosti databáz v priebehu času sa nepodarilo nájsť. Pomocou projektu *Wayback Machine*, ktorý priebežne vytvára snímky veľkej časti Internetu, je však možné napríklad zistiť, že počet záznamov v databáze ChemSpider narástol za posledné tri roky z 35 na 63 miliónov.

Ako bolo predstavené v úvode práce, jej výstupy budú využité v projekte MolGate. Pre tento zámer sú súčasné API nepostačujúce a taktiež by ich použitie vytváralo závislosť na iných systémoch, nad ktorými by nebola možná žiadna kontrola. Problémom by taktiež bola agregácia offline databáz chemických štruktúr, pre ktoré žiadne služby neexistujú. Z týchto dôvodov bol zvolený prístup, kedy vyhľadávanie bude plne v réžii MolGate. Ten má podporovať veľké množstvo rôznorodých dopytov, a práve vzhľadom na jeho užitočnosť je požadovaná aj podpora štruktúrneho vyhľadávania. Táto práca sa sústredí na dva základné typy dopytov z tejto kategórie, konkrétne vyhľadanie identickej štruktúry a podštruktúrne vyhľadávanie. V prvom prípade je cieľom nájsť záznam zodpovedajúci vzoru, a to aj v prípade, kedy bol odlišne zakreslený. Ako bolo totiž vysvetlené v kapitole 2, chemická látka môže mať viacero ekvivalentných zápisov štruktúrnym vzorcom. V druhom prípade je cieľom nájsť všetky záznamy, ktorých štruktúrny vzorec obsahuje zadaný vzor.

Existujúce databázy nemajú verejne dostupný zdrojový kód. Ak by aj mali, je pravdepodobné, že by sa dal len veľmi náročne použiť v inom informačnom systéme, než pre ktorý bol pôvodne napísaný. Práve preto sa táto práca zaoberá najskôr všeobecným návrhom systému, predstavením princípov a algoritmov, a až následne konkrétnou implementáciou. Pri implementácii sa predpokladá využitie jazyka C# a prostredia .Net, aby následná integrácia s MolGate bola čo najjednoduchšia. Taktiež je požadované, aby sa využívali dátové štruktúry, ktoré boli navrhnuté pre použitie naprieč MolGate, pretože konverzia by následne spomaľovala nadväzujúce výpočty. Cieľom je tiež dosiahnuť lepšie alebo aspoň porovnateľné výsledky, než aké ponúkajú súčasné riešenia.

Aj napriek plánovanej integrácii do MolGate je potrebné výsledný systém pre vyhľadávanie určitým spôsobom demonštrovať, čo umožní overiť funkčnosť a splnenie cieľov práce, a taktiež bude slúžiť ako návod pre následné použitie. Podobne z pohľadu diplomovej práce sa tak bude jednať o ucelené riešenie, ktoré je možné priamo použiť. Je však nutné zdôrazniť, že práca sa sústredí v prvom rade na samotné vyhľadávanie, a tak spôsob, akým bude systém demonštrovaný, nemusí byť optimálny ani vhodný pre produkčné nasadenie.

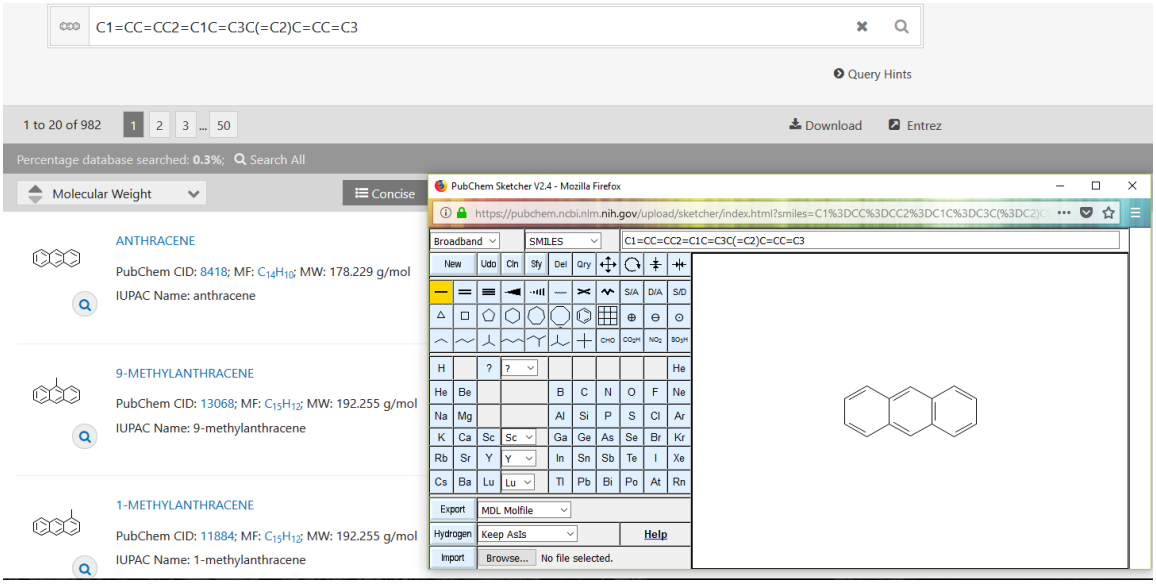
4.2 Existujúce riešenia

Existuje niekoľko online chemických databáz, ktoré podporujú štruktúrne vyhľadávanie. Medzi jednu z najväčších patrí databáza PubChem, ktorá bude podrobnejšie predstavená v sekcii 4.2.1. Ďalšími databázami hodnými povšimnutia sú napríklad ChemSpider alebo databáza ZINC. V prípade online databáz sa jedná o uzavreté systémy, ktoré poskytujú požadovanú funkcionalitu, ale sú naviazané na konkrétnu dátovú sadu a ich implementácia nie je verejne dostupná. V prípade, že pre chemickú databázu neexistuje zodpovedajúci vyhľadávací nástroj, je možné použiť chemoinformatické knižnice k jeho vytvoreniu. Tých opäť existuje veľký počet. Jednak existujú knižnice s verejne dostupnými zdrojovými kódmi, napríklad CDK, OpenBabel alebo RDKit, ale taktiež aj komerčné knižnice ako spomínaná knižnica Daylight alebo OEChem. Ich vzájomné porovnanie nie je cieľom tejto práce, ale konkrétne knižnica RDKit bola zvolená pre podrobnejší popis v časti 4.2.2 na základe jej popularity, stále aktívneho vývoja a implementácie v jazyku C++, ktorú je možné použiť z mnohých programovacích jazykov.

4.2.1 PubChem

PubChem je databáza chemických molekúl a ich biologických aktivít vytvorená americkou národnou organizáciou NCBI. Obsahuje popis a charakteristiku pre malé molekuly

s rádovo desiatkami až stovkami atómov. Obsahuje viac ako 95 miliónov záznamov, ktoré pochádzajú z viac ako 500 zdrojov [30]. Táto databáza umožňuje užívateľom vyhľadávať pomocou webového nástroja alebo využitím API. Pri použití webového nástroja si užívateľ najskôr zvolí druh vyhľadávania, ktorý chce vykonať. Podporované je vyhľadávanie identických a podobných štruktúr, podštruktúr, nadštruktúr a 3D podobnosť. Následne môže užívateľ zadať vstup pomocou CID, čo je identifikátor molekúl v PubChem databáze, SMILES, InChI alebo ho zakresliť pomocou grafického editora. Každé vyhľadávanie má taktiež množinu parametrov, ktoré je možné podľa potreby prispôsobiť, napríklad v prípade podštruktúrneho vyhľadávania nezohľadňovanie stereo informácie alebo izotopov prvkov. Po spustení vyhľadávania je o priebehu užívateľ informovaný pomocou indikátora, ktorý je aktualizovaný v pravidelných intervaloch. Po dokončení vyhľadávania sa zobrazí stránka so zoznamom nájdených záznamov. V čase písania tejto práce je dostupná aj testovacia verzia nového webového rozhrania, ktoré je responzívnejšie a priebeh vyhľadávania je pred užívateľom viac skrytý. Vyhľadávanie je spočiatku obmedzené iba na malú časť databázy a až následnou voľbou sa prehľadá aj jej zvyšok. Na obrázku 4.1 je možné vidieť časť tohto nového rozhrania spolu so štruktúrnym editorom, ktorý bol použitý pre zakreslenie vzoru.



Obr. 4.1: Nové webové rozhranie databázy PubChem spolu so štruktúrnym editorom

Druhým spôsobom vyhľadávania v tejto databáze je použitie niektorého z dostupných API [31]. Ako príklad je možné uviesť REST API, ktorého princíp spočíva vo vykonávaní operácií ako odpoveď na rôzne HTTP požiadavky na špecifickú URL adresu, v ktorej sú uvedené vstupné parametre. Napríklad operáciu podštruktúrneho vyhľadávania je možné spustiť volaním metódy GET na adresu `https://pubchem.ncbi.nlm.nih.gov/rest/pug/compound/substructure/smiles/C3=NC1=C(C=NC2=C1C=NC=C2)[N]3/XML`. Keďže sa môže jednať o potencionálne dlhú operáciu, server vráti identifikátor, ktorý je po dokončení operácie možné použiť na získanie výsledku. Pre tento účel slúži adresa `https://pubchem.ncbi.nlm.nih.gov/rest/pug/compound/listkey/xxxxx/cids/XML`, ktorá po doplnení identifikátora vyhľadávania umožňuje získať zoznam PubChem identifikátorov nájdených záznamov.

Použitie API umožňuje vytvoriť jednoduché aplikácie, ktoré využívajú PubChem pre vyhľadávanie. Tento spôsob zadávania dopytov je však limitovaný a prekročenie týchto limitov

môže mať za následok dočasné zablokovanie služby danému užívateľovi. Tieto obmedzenia sú v prípade databáze PubChem nasledovné [31]:

- Maximálne päť dopytov za sekundu.
- Maximálne 400 dopytov za minútu.
- Spoločný výpočtový čas operácií nesmie presiahnuť 300 sekúnd za minútu, čo môže nastať pri paralelnom behu dopytov.

PubChem taktiež umožňuje stiahnutie všetkých dát pomocou FTP servera[1], a práve štruktúry z tejto databázy budú použité pre inicializáciu vlastného systému a následné testovanie.

4.2.2 RDKit

RDKit je chemoinformatická knižnica poskytujúca veľké množstvo metód potrebných pri práci s dátami o chemických látkach. Jedná sa o knižnicu s verejne dostupným kódom a licenciou BSD, ktorá je aj v súčasnosti aktívne vyvíjaná jej autorom Gregom Landrumom. Knižnica je z veľkej časti implementovaná v jazyku C++, ale niektoré pokročilé vysokoúrovňové funkcie sú implementované v jazyku Python, na ktorý je knižnica primárne orientovaná. Časť implementácie, ktorá je písaná v C++, je pomocou zaobalení (angl. wrappers) taktiež sprístupnená do jazykov Java a C#.

Knižnica v prvom rade umožňuje načítať štruktúrne dáta z rôznych formátov a reprezentácií a následne umožňuje ich analýzu, modifikáciu, vykreslenie či mnohé ďalšie operácie. Z pohľadu tejto práce je však zaujímavá hlavne implementácia podštruktúrneho vyhľadávania. Pre vyhľadávanie izomorfného podgrafu je použitý algoritmus VF2, ktorý bol upravený tak, aby nezohľadňoval orientáciu hrán a pre porovnávanie vrcholov a hrán používal informácie o atómoch, respektíve väzbách. Inak sa ale jedná o pôvodnú implementáciu z knižnice VFLib. RDKit umožňuje vykonať podštruktúrne vyhľadávanie pre dve konkrétne chemické štruktúry alebo vyhľadanie vzoru v rámci kolekcie štruktúr uloženej v pamäti. Problémom je, že funkcie sprístupnené užívateľom knižnice neumožňujú zmeniť spôsob, akým sa atómy a väzby porovnávajú. Toto je často potrebné, pretože v chémii existujú situácie, pre ktoré neexistuje jedno správne riešenie alebo metóda, ale výber závisí od konkrétneho prípadu použitia. Knižnica napríklad využíva pre detekciu kružníc v grafe metódu najmenšej množiny najmenších kružníc (angl. Smallest Set of Smallest Rings, SSSR), ktorej správnosť je diskutabilná a v niektorých prípadoch môže viesť k nedeterminizmu [22]. Príznak, či je atóm súčasťou kružnice, je potom jedným z kritérií pri porovnávaní vrcholov vzorového a cieľového grafu a toto chovanie nie je možné zmeniť bez úpravy zdrojových kódov knižnice.

Podštruktúrne vyhľadávanie je taktiež dostupné vo forme zásuvného modulu pre PostgreSQL databázu. V takom prípade sú štruktúrne dáta uložené v komprimovanej forme priamo v databáze ako jednotlivé záznamy. Každý záznam má priradený odtlačok pre podštruktúrne vyhľadávanie, ktorý bude bližšie popísaný v časti 5.3.3. Nad týmito odtlačkami je vytvorený špeciálny GiST index, ktorý urýchľuje porovnávanie odtlačkov. Samotné vyhľadávanie potom prebieha priamo v procese databázového servera využitím algoritmov z knižnice RDKit. Výhodou tohto prístupu je, že umožňuje jednoducho vytvoriť všeobecnú databázu chemických látok s podporou vyhľadávania a vyhľadávanie je vďaka odtlačkom, indexovaniu a komprimácii oveľa rýchlejšie než v prípade samostatnej aplikácie. Nevýhodou

[1] Adresa FTP servera: ftp://ftp.ncbi.nlm.nih.gov/pubchem

ale je, že všetky úlohy pri vyhľadávaní preberá databázový server, ktorý je možné pri vyššom počte požiadaviek preťažiť. Škálovanie takéhoto systému nie je jednoduché, pretože štruktúrne dáta sú priamo súčasťou databázy a v prípade potreby pridania ďalšieho servera by ich bolo nutné replikovať. Jedná sa preto iba o čiastočné riešenie, ktoré môže postačovať potrebám jednotlivcov, ale v prípade väčších systémov by si vyžadovalo ďalšie podporné programy. Navyše rovnako platí, že vyhľadávanie nie je možné nijak parametrizovať a prispôsobiť vlastným potrebám.

4.3 Vytýčenie cieľov

S ohľadom na zamýšľané nasadenie systému a ostatné existujúce riešenia je vhodné zadefinovať požiadavky, ktoré by mal vytvorený systém spĺňať. V prvom rade by mal podporovať požadované operácie. Tie by mali byť implementované čo najefektívnejšie, s cieľom dokončiť vyhľadávanie v čo najkratšom čase, pretože ich výstupy budú často použité pri ďalších výpočtoch. Úlohou identického vyhľadávania bude najčastejšie identifikácia danej štruktúry a získanie zodpovedajúceho záznamu, čo sa napríklad v prípade simulácií bude využívať často. Je preto potrebné, aby vyhľadávanie skončilo maximálne do 10 ms, čo umožní aspoň 100 takýchto vyhľadávaní za sekundu. V prípade podštruktúrneho vyhľadávania je nutné prihliadnuť na náročnosť operácie a fakt, že aj v existujúcich databázach trvá táto operácia rádovo jednotky až desiatky sekúnd. Hlavným cieľom bude preto prísť s rýchlejším algoritmom, než je zaužívaný algoritmus VF2 a za referenčnú implementáciu bude považovaná tá v knižnici RDKit. Okrem toho bude cieľom optimalizovať formát, v akom sú štruktúrne dáta uložené, čo by sa malo prejaviť ďalším zrýchlením vyhľadávania. Pri návrhu by sa potom malo pamätať na to, že systém bude v budúcnosti potrebné škálovať, či už kvôli zväčšujúcemu sa objemu dát alebo nárastu počtu užívateľov. Celkovo je tieto požiadavky možné zhrnúť do nasledujúcich bodov:

1. Implementácia dvoch základných typov štruktúrneho vyhľadávania:
   (a) Identické.
   (b) Podštruktúrne.
2. Doba vyhľadania identickej molekuly nesmie presiahnuť 10 ms.
3. Implementovaný algoritmus pre podštruktúrne vyhľadávanie musí byť rýchlejší, než je implementácia VF2 v knižnici RDKit. Čas behu algoritmu pre jednu dvojicu by nemala presiahnuť 100 µs. Implementácia musí taktiež umožňovať parametrizáciu vyhľadávania.
4. Vytvorenie optimálneho formátu pre uloženie štruktúrnych dát. Doba načítania aj veľkosť záznamu musí byť výrazne menšia oproti formátu SDF.
5. Použitie dátových štruktúr spoločných pre celý projekt MolGate.
6. Implementácia v jazyku C# a využitie technológii z prostredia .Net.
7. Riešenie by malo umožňovať v prípade potreby jednoduché rozšírenie systému o ďalšie výpočtové prostriedky.
8. Výsledné riešenie musí byť plne funkčné, čo bude overené jednotkovými testami a webovou aplikáciou.

Kapitola 5

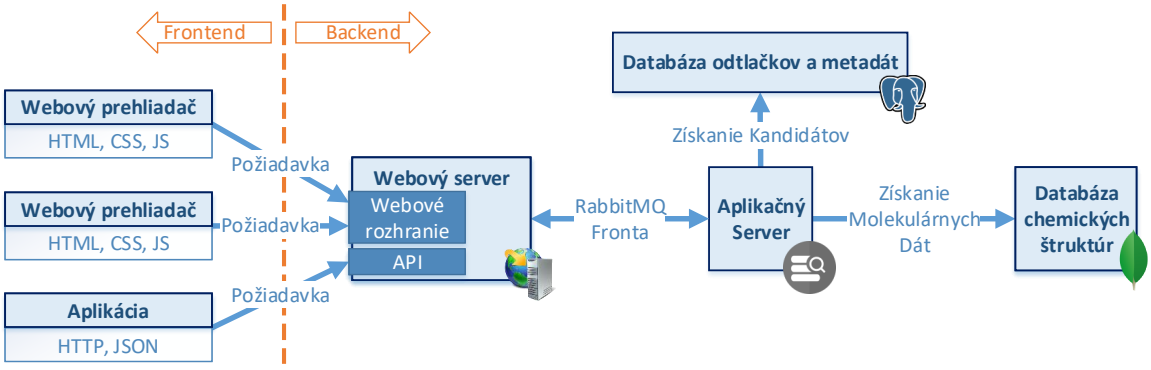
Návrh riešenia

Hlavnou témou tejto kapitoly je návrh systému pre vyhľadávanie. To zahŕňa výber technológií, návrh samotných operácií vyhľadávania, ako aj zvolenie databázy a spôsobu ukladania štruktúrnych dát v rámci systému.

Okrem návrhu samotného systému pre vyhľadávanie je nutné navrhnúť aj spôsob, akým bude jeho funkcionalita demonštrovaná. Z hľadiska zamýšľaného použitia aj čo najjednoduchšieho sprístupnenia systému užívateľom dáva najväčší zmysel vytvorenie webovej služby. Tá býva zvyčajne rozdelená na dve základné logické časti, *frontend* a *backend*. Toto rozdelenie vyplýva z klient-server architektúry, na ktorej je postavená veľká časť webových služieb. Frontend je časť služby zodpovedná za prezentáciu, čiže napríklad zadanie vstupov a zobrazenie výsledkov a prístup k zvyšným častiam systému. V prípade webovej služby sa bežne jedná o kód určený pre webový prehliadač. Backend je potom časť služby, ktorá prijíma a spracováva požiadavky od užívateľov, vykonáva požadované úlohy a informuje o ich výsledku.

Úvodom tejto kapitoly je v sekcii 5.1 predstavený už finálny návrh celého systému, aby čitateľ v prvom rade získal prehľad o jeho častiach. Zvyšné sekcie sa zaoberajú analýzou jednotlivých častí, výberom technológií a odôvodnením, prečo bol zvolený daný spôsob riešenia. Záverom je časť 5.5 venovaná dvojici prípadov použitia grafových databáz, ktoré boli zvažované pri návrhu za účelom zefektívnenia a snahy preskúmať aj iné spôsoby riešenia.

5.1 Navrhovaná architektúra systému pre vyhľadávanie



Obr. 5.1: Architektúra navrhovaného systému

Cieľom tejto časti je predstaviť celkovú architektúru systému a úlohu jeho jednotlivých častí skôr, než sa pristúpi k ich podrobnejšiemu popisu. Obrázok 5.1 znázorňuje časti systému a vzťahy medzi nimi.

Keďže bola zvolená demonštrácia formou webovej aplikácie, súčasťou výsledného systému musí byť v prvom rade webový server. Ten má za úlohu obstarávať požiadavky z webových prehliadačov a prezentovať výsledky formou webových stránok. Okrem toho tento server sprístupňuje službu vyhľadávania pomocou API, čo umožňuje jej využitie v rámci samostatných aplikácií.

Najdôležitejšou časťou systému je aplikačný server, ktorý je zodpovedný za samotné vyhľadávanie. Tento server implementuje vyhľadávacie algoritmy a spolupracuje s databázovými servermi. Tie sú v systéme dva, jeden pre uloženie odtlačkov a ďalších metadát o molekulách a druhý pre uloženie samotných štruktúrnych dát. Toto rozdelenie bude objasnené v časti 5.4.

Komunikácia medzi webovým a aplikačnými servermi prebieha pomocou komunikačného kanála, ktorý pracuje na princípe fronty. Po prijatí požiadavky na vyhľadávanie od klienta vytvorí webový server požiadavku, v ktorej uvedie parametre vyhľadávania, a vloží ju do príslušnej fronty, odkiaľ si ju aplikačný server prevezme a začne s jej spracovaním. Toto riešenie umožňuje pridávať do systému viacero aplikačných serverov podľa potreby a taktiež ich rozmiestnenie na niekoľko oddelených fyzických serverov. Výber technológie, ktorá túto komunikáciu zabezpečuje, je predmetom sekcie 5.2.3.

5.2 Webový server

Webový server má tri hlavné úlohy:

- Obsluha HTTP požiadaviek z webových prehliadačov a poskytnutie príslušnej webovej stránky.
- Obsluha HTTP požiadaviek na API z aplikácií. API umožňuje spúšťať všetky podporované operácie a dopytovať sa na ich výsledok.
- Komunikácia s aplikačnými servermi.

Najdôležitejšou voľbou z pohľadu technológií je v prípade webového servera výber frameworku. Framework je možné chápať ako sadu knižníc, odporúčaných postupov a architektonických vzorov, pomocou ktorých je možné zjednodušiť vývoj, pretože riešia typické problémy pre danú oblasť. Webových frameworkov existuje pomerne veľa, ale v mnohých prípadoch sú zviazané s konkrétnym programovacím jazykom. Napríklad pre jazyk Python existuje populárny framework Django, pre Ruby je to Ruby on Rails. Keďže je požadovaná implementácia v jazyku C#, je možné podobným spôsobom obmedziť výber len na niekoľko možností. Tými sú ASP.NET, jeho nástupca ASP.NET Core a komunitný projekt Nancy. Každý z týchto frameworkov je aktívne vyvíjaný, má rozsiahlu dokumentáciu a licencia umožňuje bezplatné použitie aj v komerčnom softvéri. ASP.NET Core a Nancy sú navyše open-source a podporujú aj iné operačné systémy než Windows. Konečné rozhodnutie použiť ASP.NET Core bolo založené na jeho perspektíve do budúcnosti, väčšej komunite a meraniach výkonnosti[1], v ktorých sa radí na popredné priečky. Najaktuálnejšia stabilná verzia tohto frameworku, ktorá bude použitá aj v práci, je verzia 2.0.

[1] Zdroj: https://www.techempower.com/benchmarks/#section=data-r15&hw=ph&test=plaintext

5.2.1 Klientská časť

Klientská časť, čiže spomínaný frontend, je v prípade ASP.NET Core pomerne zviazaný s webovým serverom, preto je jej návrh súčasťou tejto sekcie. Pre tvorbu stránok sa využíva zápis pomocou syntaxe Razor, ktorá je rozšírením značkovacieho jazyka HTML a umožňuje vkladať fragmenty kódu napísaného v jazyku C#, ktoré budú vyhodnotené serverom pred zaslaním stránky klientovi. Okrem toho budú použité bežné jazyky a technológie používané pri vývoji webových aplikácii ako JavaScript, CSS, Bootstrap a jQuery.

Z pohľadu užívateľa by mala táto časť systému umožniť zakresliť požadovaný vzor a určiť parametre vyhľadávania, minimálne teda, či sa jedná o identické alebo podštruktúrne vyhľadávanie. K tomu je potrebné poskytnúť grafický nástroj, takzvaný štruktúrny editor, ktorý slúži k zakresleniu vstupnej štruktúry. Existuje niekoľko open-source nástrojov vytvorených v jazyku JavaScript, ktoré umožňujú vloženie priamo do webovej stránky, napríklad JSME[1] alebo Ketcher[2]. V tejto práci však bude použitý súkromný editor, ktorý bol napísaný v jazyku TypeScript a transpilovaný na JavaScript aplikáciu. Tento editor bude neskôr taktiež súčasťou projektu MolGate, preto je žiadúce jeho otestovanie v kombinácii s vyhľadávaním. V každom prípade poskytujú všetky štruktúrne editory veľmi podobnú funkcionalitu a nemal by byť preto problém zvolený editor vymeniť za iný.

Zakreslenie vzoru a zadanie parametrov sa bude diať na úvodnej stránke. Po spustení vyhľadávania bude užívateľ presmerovaný na stránku s výsledkami, kde bude informovaný o priebehu operácie a po dokončení vyhľadávania sa mu zobrazia nájdené záznamy formou tabuľky s grafickým náhľadom daných štruktúr.

Vzhľadom na plánované vytvorenie API, vyhľadávanie môže byť priamo v réžii webového prehliadača, ktorý pomocou asynchrónnych požiadaviek spustí vyhľadávanie a bude sa periodicky dopytovať na jeho výsledok. Tým sa celkovo zjednoduší implementácia a dôjde k odbremeneniu webového rozhrania, pretože samotnú webovú stránku nebude nutné neustále aktualizovať.

5.2.2 Aplikačné rozhranie

Pri vytváraní aplikačného rozhrania existujú dve najrozšírenejšie metódy, ako ho sprístupniť. SOAP (angl. Simple Object Access Protocol) je protokol, ktorý bol vytvorený pre spoločnosť Microsoft a je založený na výmene správ vo formáte XML cez sieť. Na prenos správ sa najčastejšie využíva protokol HTTP, no podporované sú aj iné protokoly, napríklad SMTP. SOAP definuje formát správ, kde sa každá správa skladá z hlavičky a tela zaobalených v spoločnej obálke. Okrem toho sú definované takzvané šablóny komunikácie, z ktorých najznámejšia je RPC šablóna pre vytvorenie architektúry typu klient-server [35].

Druhou možnosťou je použitie REST (angl. Representational State Transfer) architektúry, ktorá je založená na prístupe k rôznym zdrojom identifikovaných pomocou URI (angl. Unique Resource Identifier) využitím protokolu HTTP. Túto architektúru popísal Roy Fielding vo svojej dizertačnej práci [14]. Jednotlivé zdroje predstavujú napríklad súbory, záznamy alebo aj algoritmy. Metódy protokolu HTTP potom slúžia k manipulácii s týmito zdrojmi a zvyknú mať nasledovnú sémantiku:

- GET – používaná k získaniu reprezentácie zdroja.
- POST – používaná k vytvoreniu nového zdroja.

[1] Odkaz: http://peter-ertl.com/jsme/
[2] Odkaz: http://lifescience.opensource.epam.com/ketcher/

- DELETE – používaná k odstráneniu existujúceho zdroja.
- PUT – používaná k vytvoreniu alebo aktualizácii zdroja v prípade jeho existencie.

REST je primárne určený pre architektúru klient-server a nešpecifikuje formát správ, ktorý tak môže byť JSON, XML alebo ľubovoľný iný. Tento princíp umožňuje voľnejšiu väzbu medzi serverom a klientom, pretože neexistuje toľko predpokladov, ktoré musia byť splnené, ako pri protokole SOAP.

Rozhodnutie medzi SOAP a REST je pomerne náročné, keďže sa jedná o veľmi odlišné prístupy. Ako nevýhoda SOAP sa často spomína práve naviazanosť na XML formát, ktorý vytvára dlhé správy a je taktiež náročnejší na spracovanie. Naopak jeho výhody, ako integrované zabezpečenie správ a autentifikácia, nie sú z pohľadu tejto práce zaujímavé. Pretože sú webové REST rozhrania všeobecne jednoduchšie na vytvorenie a v dnešnej dobe veľmi rozšírené[1], bolo aj pre túto prácu rozhodnuté vytvoriť API pomocou architektúry REST a správy budú vo formáte JSON.

URI	Metóda	Popis
`api/search/`	`POST`	Spustí nové vyhľadávanie.
`api/search/{id}`	`GET`	Získa výsledok vyhľadávania.
`api/search/progress/{id}`	`GET`	Získa informácie o priebehu vyhľadávania.
`api/molecule/{mol-id}`	`GET`	Získa dáta pre jednu molekulu.

Tabuľka 5.1: Prehľad operácií poskytovaných aplikačným rozhraním.

Výsledné API bude v prípade navrhovaného systému pomerne jednoduché. Najdôležitejším zdrojom je samotné vyhľadávanie. Je potrebné mať možnosť vytvoriť nové vyhľadávanie, získať informácie o jeho priebehu a po ukončení operácie prevziať množinu výsledkov. Poslednou požadovanou operáciou je získanie dát o molekule podľa jej identifikátora, čo umožní zobrazenie výsledkov po dokončení vyhľadávania. Tabuľka 5.1 uvádza jednotlivé URI a metódy, ktoré umožňujú vykonať tieto operácie.

5.2.3 Komunikácia s aplikačným serverom

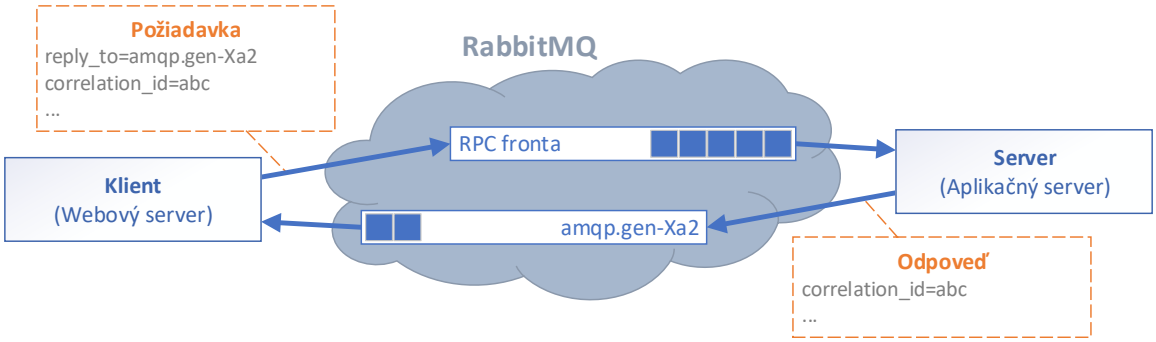
Medzi webovým a aplikačným serverom je nutné vytvoriť spoľahlivý komunikačný kanál, ktorý bude slúžiť pre zasielanie príkazov na spustenie vyhľadávania a spätné informovanie o priebehu a výsledkoch. Spôsob, akým bude komunikácia zabezpečená, by mal podporovať dynamické pridávanie a odoberanie aplikačných serverov, aby mohol byť systém v prípade potreby rozšírený o ďalšie inštancie serverov. Dôležité je, aby tento spôsob komunikácie umožňoval zasielanie správ cez sieť, teda aj na iné fyzické servery, než na akom je spustený webový server. Vďaka tomu bude možné do systému pridávať ďalšie výpočtové prostriedky. Keďže sa v hrubej podstate jedná o predanie požiadaviek od klientov aplikačným serverom, bude aj táto komunikácia založená primárne na komunikačnom modeli požiadavka-odpoveď.

Najpriamočiarejší spôsob, ako takúto komunikáciu zabezpečiť, by bola priama komunikácia medzi servermi, napríklad pomocou protokolu TCP/IP. Aplikačný server by vytvoril socket, na ktorom by čakal na nové spojenia a po prijatí požiadavky od webového servera by komunikácia prebiehala na tomto spojení až do ukončenia operácie. Problémov s týmto prístupom je niekoľko. V prvom rade je nutné vytvoriť aplikačný protokol, pomocou ktorého

[1] Zdroj: https://blog.restcase.com/the-rise-of-rest-api/

budú jednotlivé strany komunikovať. Okrem toho by museli účastníci komunikácie navzájom poznať svoje IP adresy. Taktiež by sa muselo určitým spôsobom riešiť vyvažovanie záťaže, čiže rozdelenie požiadaviek medzi dostupné aplikačné servery.

Namiesto vytvárania vlastného riešenia v tejto oblasti je rozumnejšie použiť existujúce protokoly, ktoré mnohé z týchto problémov riešia. Zasielanie správ (angl. messaging) je koncept, ktorý umožňuje lepšie oddeliť zasielateľov správ od ich prijímateľov tým, že jednotkou informácie sa stáva správa, ktorá je doručená na určitý kanál, odkiaľ si ju záujemca môže prevziať. Protokolov pre zasielanie správ existuje niekoľko, medzi najznámejšie patria AMQP, MQTT a STOMP. Každý z týchto protokolov má svoje prípady použitia, no v prípade zamýšľaného distribuovaného systému sa ako najlepší javí práve AMQP, ktorý používa binárny protokol na prenos správ a jeho hlavnými prednosťami sú spoľahlivosť a interoperabilita [24]. V každom prípade nie je cieľom práce implementovať takýto protokol, preto bude použitý niektorý z existujúcich sprostredkovateľov správ (angl. message broker), ktorý tento protokol implementuje. Sprostredkovateľ správ je samostatná entita, ktorá poskytuje funkcionalitu pre doručovanie správ pomocou niektorého z existujúcich protokolov. Pre AMQP existuje hneď niekoľko sprostredkovateľov správ, napríklad ActiveMQ, RabbitMQ alebo ZeroMQ. Na základe porovnania sprostredkovateľov a dostupných knižníc pre ich použitie z jazyka C# bola nakoniec zvolená kombinácia RabbitMQ s knižnicou EasyNetQ. Cieľom bolo v prvom rade zvoliť riešenie, ktoré bude umožňovať jednoduchú implementáciu a podporovať požadovaný model komunikácie.



Obr. 5.2: Implementácia komunikačného modelu požiadavka-odpoveď v rámci RabbitMQ

Komunikácia s využitím RabbitMQ bude prebiehať nasledovne. Po prijatí požiadavky na vyhľadávanie od užívateľa alebo aplikácie bude táto vložená do fronty spoločnej pre všetky aplikačné servery, odkiaľ si ju aplikačný server prevezme k spracovaniu. Ak bude dostupných viacero aplikačných serverov, vo východzom nastavení budú požiadavky rozdelené na princípe round-robin. Server následne odpovie identifikátorom vyhľadávania, tak ako je to uvedené v špecifikácii API, ale navyše identifikuje sám seba pomocou mena, ktoré mu bolo priradené pri spustení. Toto meno musí byť v rámci systému unikátne, pretože sú od neho odvodené názvy komunikačných front, ktoré webový server použije pre dopytovanie na priebeh a výsledok operácie. Vďaka vytvoreniu unikátnych front pre každý aplikačný server bude možné spätne kontaktovať ten, ktorý je zodpovedný za konkrétne vyhľadávanie. Je nutné podotknúť, že pre zaslanie požiadavky a čakanie na odpoveď, čo je v terminológii RabbitMQ označované ako vzdialené volanie procedúr, sa v skutočnosti vytvoria dve fronty, jedna do ktorej sa vloží požiadavka a druhá slúži pre uloženie odpovede. Tento princíp znázorňuje obrázok 5.2.

5.3 Aplikačný server

Aplikačný server implementuje operácie popísané v časti 5.2.2. V prvom rade teda musí podporovať samotné operácie vyhľadávania, ktoré sú popísané v nasledujúcich častiach 5.3.1 a 5.3.2. Okrem toho musí implementovať rozhranie pre komunikáciu s databázovými servermi, ktorých voľba je zdôvodnená v časti 5.4. K sprístupneniu relačnej databázy bude použitá knižnica Entity Framework pre objektovo relačné mapovanie (ORM), ktoré zabezpečuje mapovanie tabuliek a procedúr v databáze na triedy a funkcie v prostredí .Net. Okrem jednoduchšej práce s databázou bude mať použitie ORM výhodu, že zvolenú relačnú databázu bude možné v prípade potreby pomerne jednoducho vymeniť za inú.

Server sa po spustení prihlási k odberu správ z RabbitMQ fronty, do ktorej sú vkladané požiadavky na vyhľadávanie. Po obdržaní požiadavky vytvorí novú vyhľadávaciu úlohu a priradí jej číselný identifikátor. Každý server bude musieť mať v rámci systému unikátny názov, aby bolo možné jednoznačne určiť, ktorý server spracováva danú úlohu. Priamu odpoveď na požiadavku bude potom tvoriť práve názov servera a identifikátor úlohy. Okrem toho sa server zaviaže odpovedať na požiadavky na priebeh a výsledok operácie, pre ktoré budú vytvorené samostatné fronty s názvom odvodeným od názvu servera. Po dokončení úlohy bude jej výsledok, zoznam identifikátorov nájdených záznamov, po istú dobu uložený na serveri, odkiaľ si ho môže klient prevziať. Po uplynutí tejto doby bude výsledok zmazaný, aby proces nespotreboval príliš veľa pamäte. V prípade potreby uchovávať výsledok po dlhšiu dobu by bolo možné uložiť ho do databázy, odkiaľ by bol v prípade zmazania v pamäti znova získaný.

5.3.1 Vyhľadanie identickej štruktúry

Ako bolo uvedené v sekcii 4.3, tento typ vyhľadávania má za cieľ nájsť chemicky ekvivalentnú molekulu na základe štruktúry. Užívateľ teda nemusí zadať molekulu presne v takom tvare, v akom je uložená v databáze, napríklad môže molekulu inak priestorovo zakresliť alebo inak zaznačiť aromatické väzby (viď rezonančné formy v časti 2.2.5). Systém by mal byť aj napriek tomu schopný nájsť zodpovedajúci záznam v databáze a poskytnúť informácie o molekule.

Z tohto dôvodu nie je možné použiť primitívny algoritmus, v ktorom by sa porovnávali molekuly na základe atómov a väzieb tak, ako sú uložené v dátovej štruktúre. Nielenže by takýto algoritmus nedokázal rozpoznať chemicky ekvivalentné molekuly, ale stačilo by, aby boli porovnávané prvky uložené v inom poradí, a algoritmus by záznam nenašiel. Tieto problémy by bolo možné riešiť využitím grafových algoritmov, konkrétne grafového izomorfizmu, kedy by sa hľadalo priradenie atómov hľadanej molekuly na atómy molekuly uloženej v databáze. Popri bežnom grafovom izomorfizme by ale algoritmus musel uvažovať aj spomínané chemické aspekty. K tomuto účelu by bolo možné využiť algoritmus pre hľadanie izomorfného podgrafu, ktorý už bude implementovaný v rámci podštruktúrneho vyhľadávania. Keďže sa však jedná o hľadanie podgrafu, museli by byť dodatočne aplikované ešte dve kritériá. Prvým je, že obe molekuly musia mať zhodný počet atómov, ktoré by mohlo byť aplikované ešte pred samotným algoritmom. Ako ďalšie je potrebné kontrolovať, či je počet väzieb na priradených atómoch zhodný. Toto kritérium vyplýva z toho, že podštruktúrny algoritmus hľadá neindukovaný podgrafový izomorfizmus, kedy podmienkou pre priradenie atómov je existencia väzby vo vzore. Naopak algoritmus nekontroluje neexistenciu väzby, čo by mohlo mať za následok, že nájdená molekula obsahuje väzby, ktoré vo vyhľadávanom vzore nie sú. Výhodou práve uvedeného prístupu je jednoduchá realizácia, ak už je imple-

mentovaný algoritmus pre podštruktúrne vyhľadávanie. Vzhľadom na požiadavku rýchleho identického vyhľadávania však tento spôsob nepripadá v úvahu.

Lepším riešením je reprezentovať molekulu pomocou jednoznačného a univerzálneho textového identifikátora, akým je napríklad InChI, resp. pomocou skrátenej formy InChIKey. Ich predstaveniu bola venovaná sekcia 3.2.2. Univerzálnosť InChI zaručuje, že chemicky ekvivalentné molekuly budú mať vždy rovnaký identifikátor. V opačnom prípade by musela databáza ukladať pre danú molekulu všetky rôzne identifikátory. V tomto prípade je však dôležitejšia jednoznačnosť, vďaka čomu je možné zaručiť, že v prípade zhody InChI sa skutočne jedná o hľadanú molekulu. Je treba pripomenúť, že InChIKey túto vlastnosť nemá, preto pri nájdení zhodných InChIKey je nutné dodatočne porovnať aj InChI.

Takýto textový reťazec je už možné bežným spôsobom vložiť do databázy, indexovať a následne rýchlo vyhľadať. K vytvoreniu identifikátora je možné použiť referenčnú knižnicu dostupnú online[1]. Tá umožňuje prevod chemickej štruktúry na InChI a taktiež prevod InChI na InChIKey. Knižnica používa vlastné dátové typy a navyše je písaná v jazyku C, preto bude nutné vytvoriť zaobalujúcu knižnicu, ktorá sprístupní jej funkcionalitu.

5.3.2 Podštruktúrne vyhľadávanie

Pri tomto type vyhľadávania je cieľom nájsť všetky chemické štruktúry v databáze, ktoré obsahujú zadaný vzor ako podgraf. Na rozdiel od predchádzajúceho typu vyhľadávania nie je v tomto prípade známa žiadna vhodná alternatívna reprezentácia, ktorá by umožňovala vyhnúť sa použitiu grafových algoritmov.

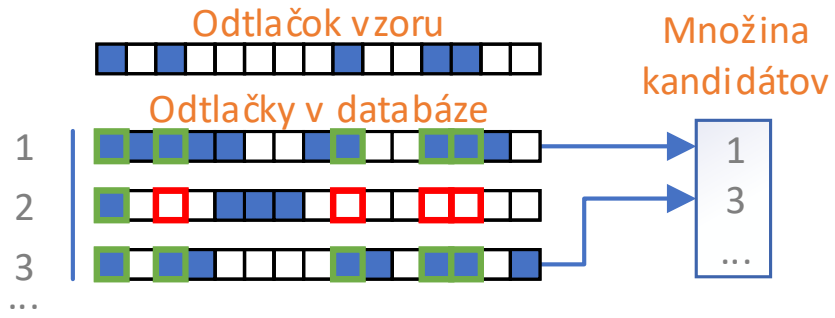
V prvom rade je preto nutné vybrať niektorý z algoritmov pre hľadanie izomorfného podgrafu predstavených v sekcii 3.4. Z uvedenej štvorice pripadajú do úvahy všetky, okrem Ullmannovho algoritmu. V danej sekcii bolo totiž uvedené, že VF2 je priamo v kontexte chemických štruktúr efektívnejší než Ullmannov algoritmus a RI by mal dosahovať ešte lepšie výsledky. Glasgow potom taktiež dosahoval od určitého trvania behu algoritmu lepšie výsledky než VF2, ale tu už sa jednalo o porovnanie na sade všeobecných grafov. Nie je tak vopred zrejmé, ktorý z týchto troch algoritmov bude v prípade podštruktúrneho vyhľadávania lepšou voľbou. Zvoleným riešením je preto algoritmy implementovať, vykonať ich porovnanie nad reprezentatívnou sadou dát a na základe výsledkov sa rozhodnúť, ktorý algoritmus má byť použitý.

Základný algoritmus umožňuje zistiť, či sa vzorový graf nachádza v nejakom inom grafe ako podgraf. K tomu stačí nájsť jednoznačné priradenie vrcholov, v ktorom medzi vrcholmi podgrafu budú všetky tie hrany, ktoré sú aj vo vzorovom grafe. U chemických štruktúr ale nie sú všetky vrcholy a hrany ekvivalentné. Atómy majú atribúty, ako napríklad prvok alebo izotop, a väzby majú rôznu násobnosť. Vo všeobecnosti je žiadané, aby priradenie mohlo vzniknúť iba ak sa všetky atribúty zhodujú. V niektorých prípadoch však môže byť užitočné niektoré z nich ignorovať. To už väčšinou závisí od užívateľa a toho, čo sa snaží nájsť. Je preto žiadané mať možnosť vyhľadávanie parametrizovať. Vhodným riešením je upraviť algoritmus tak, aby umožňoval použitie ľubovoľného rozhodovacieho algoritmu pri porovnávaní vrcholov a hrán. To je možné napríklad predaním dvoch funkcií algoritmu, ktorých vstupom sú dva vrcholy, resp. hrany a výstupom je pravdivostná hodnota, či je priradenie možné. Takýmto spôsobom bude možné vyhľadávanie parametrizovať, zatiaľ čo telo algoritmu bude vo všetkých prípadoch rovnaké.

Ako bolo spomenuté v sekcii 4.3, aj veľmi rýchly algoritmus by veľkú databázu prehľadával pridlho. Zrýchľovanie samotného algoritmu nie je riešením, pretože by sa dosiahlo

[1] Odkaz na stiahnutie: http://www.inchi-trust.org/download/105/INCHI-1-SRC.zip

bodu, kedy bude limitujúce samotné načítanie štruktúrnych dát z databázy. Je preto nutné obmedziť množinu prehľadávaných záznamov. Navrhovaným riešením je použitie molekulárnych odtlačkov, ktoré boli popísané v sekcii 3.3. Všeobecne sa jedná o bitové polia, ktoré hrubo popisujú štruktúru molekuly. Vďaka nim je možné vylúčiť záznamy, ktoré zjavne nemôžu obsahovať daný vzor, pretože im chýbajú určité charakteristické črty. Porovnaním odtlačkov vzoru a záznamov v databáze vznikne množina kandidátov, ktorá ja obvykle oveľa menšia, než celá databáza. Odtlačky sú však zväčša používané pre odhadnutie podobnosti dvoch molekúl. Ak majú slúžiť na predvýber v rámci podštruktúrneho vyhľadávania, musia byť špeciálne pre tento účel navrhnuté. Zhodnoteniu odtlačkov z tohto hľadiska sa venuje nasledujúca časť 5.3.3.



Obr. 5.3: Princíp získania množiny kandidátov pomocou odtlačku

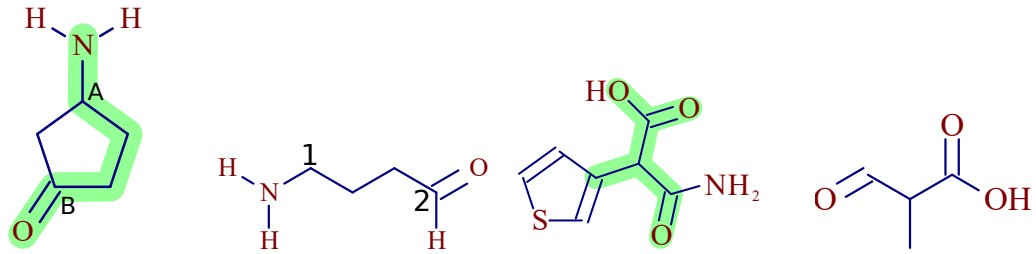
Princíp získania kandidátov je možné vidieť na obrázku 5.3. Záznam môže patriť do množiny kandidátov iba v prípade, ak majú všetky bity s hodnotou 1 v odtlačku vzoru rovnakú hodnotu aj v odtlačku pre štruktúru v databáze. V opačnom prípade by niektorá charakteristika vzoru nebola v danej štruktúre prítomná. Operáciu porovnania odtlačkov je nutné implementovať priamo v databáze, aby sa zamedzilo spomínanému problému so zdĺhavým prenosom veľkého množstva dát.

5.3.3 Výber molekulárnych odtlačkov

V časti 3.3 bolo predstavených niekoľko významných zástupcov molekulárnych odtlačkov. Nie všetky sú však vhodné pre použitie v rámci podštruktúrneho vyhľadávania. Cieľom tejto časti je vysvetliť, prečo tomu tak je, a vybrať vhodné odtlačky pre tento účel.

Všeobecne je možné problémy s odtlačkami rozdeliť do dvoch kategórií. Tou prvou je, že odtlačok už zo svojho princípu odvodenia nemôže slúžiť pre výber kandidátov. Do tejto kategórie spadajú napríklad atómové páry alebo kruhové odtlačky. Odtlačky patriace do druhej kategórie majú predpoklad pre to, aby mohli na tento účel slúžiť, no niektoré zakódované informácie alebo rozhodnutia pri implementácii to znemožňujú. Tak je tomu napríklad pri odtlačku MACCS, ktorý pracuje na podobnom princípe ako CACTVS, alebo v prípade odtlačkov typu Daylight, ktoré je možné nájsť v knižnici RDKit pod názvom RDKit Fingerprint alebo v knižnici OpenBabel ako FP2.

K objasneniu prvej kategórie je možné použiť spomínané atómové páry. Problematickým prvkom tejto metódy je, že odtlačok je založený na kódovaní najkratšej cesty. Ako príklad uvažujme molekulu na obrázku 5.4a a hľadaný vzor 5.4b. Je vidno, že molekula daný vzor obsahuje a jeho výskyt bol zvýraznený. Uvažujme teraz dvojicu atómov (1, 2) vo vzore, ktoré sa zobrazia na atómy (A, B). Dĺžka cesty medzi atómami vo vzore je tri, zatiaľ čo v molekule existuje kratšia cesta o dĺžke dva. Tým pádom bude kódovanie tohto páru



(a) Molekula (b) Vzor (c) Molekula (d) Vzor

Obr. 5.4: Obrázok zobrazuje výskyt rôznych vzorov v molekulách. Vzor (b) je možné nájsť v molekule (a) a vzor (d) v molekule (c).

vo vzore a v molekule odlišné a bity v odtlačkoch budú mať rôzne hodnoty. Daná molekula by tak testu nevyhovela. Podobný problém nastáva u kruhových odtlačkov, ktoré kódujú okolie atómu, ale toto okolie sa vo vzore nevyskytuje, aj keď je vzor možné nájsť ako podgraf.

Odtlačok MACCS bol pôvodne vyvinutý ako podštruktúrny odtlačok, no na rozdiel od odtlačku CACTVS nie je verejne dostupný význam jednotlivých bitov. To nakoniec viedlo k miernym odlišnostiam medzi implementáciami. Navyše sa začal tento odtlačok používať aj pre vyhľadávanie podobných molekúl. Následné optimalizácie pre túto úlohu potom mohli viesť k pozmeneniu významu niektorých bitov. Spolu tieto dva problémy viedli k tomu, že mnoho implementácii MACCS nie je možné použiť pre podštruktúrne vyhľadávanie. Tak je tomu aj v prípade implementácie knižnicou RDKit. Ako príklad je možné uviesť bit 139, ktorý značí výskyt skupiny `OH`. Pre vzor popísaný pomocou SMILES ako `CC(C=O)C(=O)O` bude mať tento bit hodnotu 1, zatiaľ čo v molekule `CCC(C(=O)OCC)C(=O)OCC` hodnotu 0. Napriek tomu sa vzor v molekule vyskytuje. Problémom v tomto konkrétnom prípade sú skupiny atómov s vodíkmi, ktoré sa zakódujú, aj keď boli explicitne z molekuly odstránené. U odtlačku RDKit je potom problém, že obsahuje informácie o aromaticite. Prečo toto môže byť problematické zobrazuje dvojica obrázkov 5.4c a 5.4d. Vzor sa v molekule jednoznačne vyskytuje, no pretože jeden atóm je v molekule súčasťou aromatického cyklu, všetky cesty obsahujúce tento atóm budú zakódované odlišne, než v prípade vzoru. To spôsobí, že molekula nebude zahrnutá do kandidátnej množiny.

Problémy jednotlivých odtlačkov boli overované experimentálne pomocou jednoduchého skriptu, vďaka čomu bolo možné v prechádzajúcich odsekoch uviesť konkrétne problematické dvojice. Z testovaných odtlačkov naprieč tromi knižnicami sa nakoniec ukázal ako vhodný pre podštruktúrne vyhľadávanie jedine odtlačok PatternFingerprint z knižnice RDKit, ktorý je použitý v tejto práci. Pre všetky ostatné odtlačky bolo možné nájsť prípady, kedy odtlačok vylúčil niektoré molekuly, ktoré mali byť súčasťou množiny kandidátov. Prvotným zámerom bolo pritom použiť odtlačok CACTVS, ktorý používa aj databáza PubChem, no tento odtlačok sa nepodarilo nájsť implementovaný v žiadnej z verejne dostupných knižníc a jeho implementácia sa ukázala byť náročná, čo dokazuje internetový príspevok Andrewa Dalkeho [7]. Jeho implementácia má navyše tiež problém s vynechávaním niektorých kandidátov [19].

Zvolený odtlačok pracuje na podobnom princípe ako odtlačky typu Daylight, no kladie dôraz na to, aby sa nepoužili vzory ani nekódovali vlastnosti, ktoré by viedli k vynechaniu kandidátov. Podrobnosti implementácie je možné nájsť v časti 6.2.3. Presnosť odtlačku podľa meraní prezentovaných v internetovom príspevku [19] dosahuje 60 %. Tento údaj je možné interpretovať tak, že iba 40 % záznamov patriacich do kandidátnej množiny v skutoč-

nosti daný vzor neobsahuje. Presnosť odtlačku sa môže zdať nízka, no je nutné si uvedomiť jej dôsledky. Uvažujme databázu s desiatimi miliónmi záznamov a vyhľadávaný vzor, ktorý je možné reálne nájsť ako podštruktúru v 6000 záznamoch. Po započítaní presnosti by bola veľkosť množiny kandidátov 10000 záznamov, ktoré je nutné skontrolovať algoritmom. Ak by nebol použitý odtlačok, bolo by nutné prehľadávať celú databázu. Použitie odtlačku zredukovalo počet prehľadávaných štruktúr o 99.9 %. V takom prípade by aj pomerne zlý odtlačok s presnosťou iba 5 % dokázal výrazne zmenšiť prehľadávanú množinu.

5.4 Databáza a úložisko štruktúrnych dát

Táto sekcia sa zaoberá návrhom uloženia dát v systéme a výberom technológií, ktoré umožňujú efektívnu implementáciu. Najdôležitejšími dátami sú samotné štruktúrne dáta molekúl a metadáta ako odtlačky, InChI a InChIkey, a prípadné ďalšie informácie ako celková hmotnosť molekuly alebo odkaz na pôvodný zdroj dát. V rámci práce bolo rozhodnuté rozdeliť štruktúrne dáta a metadáta do samostatných databázových systémov. Toto rozdelenie má svoje opodstatnenie. K štruktúrnym dátam sa bude pristupovať veľmi často a bude potrebné ich počas vyhľadávania získať čo najrýchlejšie. To kladie veľké nároky na systém, ktorý bude tieto dáta ukladať. Bežné relačné databázy na tento účel nie sú príliš vhodné a ako bude uvedené v časti 5.4.1, medzi zvažovanými prístupmi bolo uloženie dát priamo na disk vo forme súborov a uloženie do NoSQL databázy. Naopak, ako bolo vysvetlené v časti 5.3.2, vyhľadanie podľa InChIKey alebo určenie kandidátov pre podštruktúrne vyhľadávanie sa bude musieť diať priamo v databáze spravujúcej tieto dáta, aby sa zamedzilo ich zdĺhavému prenosu na aplikačný server, kde by museli byť v opačnom prípade spracovávané. K efektívnej implementácii bude nutné použiť pokročilé indexy a uložené procedúry. Tieto mechanizmy sú prevažne výsadou relačných databáz.

5.4.1 Uloženie štruktúrnych dát

Štruktúrne dáta popisujú štruktúru molekuly, čiže jednotlivé atómy, väzby medzi nimi a príslušné vlastnosti. Tieto dáta budú v rámci navrhovaného systému potrebné pri podštruktúrnom vyhľadávaní. Po určení množiny kandidátov bude nutné pre tieto záznamy načítať štruktúrne dáta, aby mohol byť výskyt podštruktúry skontrolovaný algoritmom. Povaha dát je statická, čiže po vložení do systému sa nebudú meniť a vo veľkej miere budú prevládať operácie čítania.

V ideálnom prípade by boli tieto dáta vždy dostupné v pamäti, no s tým súvisia dva problémy. Dátová štruktúra, ktorá sa používa pre reprezentáciu molekuly vo zvyšku aplikácie, je optimalizovaná na výkon a jednoduchosť použitia, a preto má pomerne veľkú pamäťovú náročnosť. Príkladom je redundancia, kedy každý atóm ukladá zoznam svojich väzieb ale aj susedov. Po spriemerovaní dát pol milióna záznamov dosahuje veľkosť tejto dátovej štruktúry približne 2.5 kB. V prípade tejto reprezentácie by tak server už pre niekoľko miliónov záznamov potreboval gigabajty pamäte. Preto bolo potrebné vytvoriť reprezentáciu, ktorá je menej pamäťovo náročná. Jej popisom sa zaoberá časť 6.1.3.

V konečnom dôsledku aj pri veľmi úspornej reprezentácii sú pri desiatkach miliónov záznamov nároky na pamäť vysoké. Dnes už bežne servery majú dostupných aj niekoľko desiatok gigabajtov pamäte, no je nutné si uvedomiť, že sa jedná o jednoúčelové dáta. Ak by preto existovala možnosť pre rýchle načítanie týchto dát napríklad z disku, tak by bolo toto riešenie výrazne ekonomickejšie. Prvotným nápadom bolo použiť pamäť ako cache pre štruktúry, ku ktorým sa pristupuje najčastejšie, a zvyšok dát načítať iba v prípade

potreby. Problémom je, že sa jedná o viac-menej náhodný prístup, pretože každý vzor vytvorí inú množinu kandidátov. Vytvorenie takejto cache by preto nemalo veľký zmysel. Zostáva sa teda zamerať na čo najrýchlejšie načítanie dát do pamäte. Zvažované boli dve možnosti. Prvou bolo uloženie štruktúrnych dát priamo v súborovom systéme, kde jednému záznamu zodpovedá jeden súbor. Tou druhou bolo uloženie do databázy optimalizovanej pre získavanie hodnoty podľa kľúča.

Uloženie do súborového systému predstavuje riešenie, ktorého výhodami sú nízka réžia a možnosť priameho prístupu k dátam iba za pomoci operačného systému. Nie je potrebné vytvárať spojenia ani komunikovať s iným procesom. Nastáva však problém pri použití viacerých aplikačných serverov, ktoré by nezdieľali rovnaký hardvér. V takom prípade by bolo nutné dáta replikovať, čo by sa však stále dalo v prípade výrazne vyššej rýchlosti načítania dát odôvodniť. Vzhľadom na ostatné technológie použité pre implementáciu aplikačného servera prichádzal do úvahy iba súborový systém NTFS.

Databázových systémov založených na modely kľúč-hodnota existuje hneď niekoľko, no vo väčšine prípadov sa jedná o systémy ukladajúce dáta iba v pamäti, čo nerieši pôvodný problém. Príkladmi takýchto systémov sú napríklad Redit alebo Memcached. Následne bol zvažovaný systém Riak KV, no kvôli chýbajúcej podpore pre Windows bol nakoniec zvolený databázový systém MongoDB. Jedná sa o populárny databázový systém založený na dokumentovo orientovanom modeli [36]. Každý dokument sa skladá z množiny párov kľúč-hodnota, ktoré je možné do seba aj vnárať. Zaujímavou vlastnosťou niektorých databázových systémov, ktorú podporuje aj MongoDB, je operácia nazvaná trieštenie (angl. sharding), ktorá umožňuje rozdeliť jednu kolekciu medzi viacero databázových serverov. Vďaka tomu je možné jednoducho rozložiť záťaž medzi viacero fyzických serverov.

Podobne ako v prípade výberu algoritmu pre hľadanie izomorfného podgrafu, ani v tomto prípade nebolo vopred zrejmé, ktorý prístup je lepší. Preto boli podstatné metódy implementované pre oba prístupy a výber bol uskutočnený na základe meraní v sekcii 7.2.

5.4.2 Uloženie metadát

Najdôležitejšími metadátami o molekulách sú InChI, InChIKey a molekulárny odtlačok, ktoré sú využívané pri samotnom vyhľadávaní. Každý záznam obsahuje navyše ešte identifikátor, pomocou ktorého je možné získať príslušné štruktúrne dáta, a identifikátor zo systému PubChem, ktorý umožňuje vytvoriť odkaz na pôvodný záznam. Aby bolo možné v InChIKey vyhľadávať podľa jednotlivých blokov, je rozdelený na tri samostatné polia. Navyše údaj o tom, či sa jedná o štandardné InChI je vyňatý z druhej vrstvy a uložený taktiež v samostatnom poli. Pri výbere databázového systému bolo dôležité, aby spĺňal nasledujúce kritéria:

- Podpora dátového typu reprezentujúceho polia bajtov. Predovšetkým možnosť vykonávať bitové operácie a vytvárať indexy.
- Podpora pre kompilované zásuvné moduly. Nutné pre vytvorenie efektívnych vstavaných procedúr.
- Podpora pre Entity Framework. Musí existovať príslušný ovládač.

Na základe týchto kritérií bol zvolený databázový systém PostgreSQL. V prvom rade umožňuje vytvorenie dynamickej knižnice v jazyku C, ktorú je možné následne pridať do konkrétnej databázy ako rozšírenie a využívať jej funkcionalitu. Ďalej je možné pracovať

priamo s dátovými typmi databázy, alokovať pamäť spravovanú databázovým procesom a vytvárať vlastné agregačné funkcie a metódy indexovania.

Ďalšou vhodnou voľbou by mohol byť Microsoft SQL Server, ktorý je možné jednoducho integrovať so zvyškom vybraných technológii a taktiež umožňuje vytvoriť dynamické knižnice na platforme CLR. PostgreSQL bol uprednostnený vzhľadom na jeho open-source licenciu a podporu aj iných operačných systémov než Windows, čo môže v prípade nasadenia databázového systému na samostatný server a použitia linuxovej distribúcie viesť k nižším hardvérovým požiadavkám.

5.5 Využiteľnosť grafových databáz

Pri návrhu celkového riešenia bolo zvažované použitie grafových databáz, predstavených v časti 3.5, vďaka ich schopnosti prirodzene pracovať s grafmi, čiže napríklad aj chemickými štruktúrami. V rámci navrhovaného systému by prichádzali do úvahy dva rôzne prípady použitia, ktoré sú v nasledujúcich častiach predstavené a zhodnotené.

Využitie grafových algoritmov pri vyhľadávaní
Prvou, priamočiarou aplikáciou grafových databáz by mohlo byť využitie ich grafového modelu a algoritmov pre implementáciu požadovaných operácií. Zamýšľané použitie bolo hlavne pri hľadaní izomorfného podgfrafu, preto bude nasledujúci text venovaný preskúmaniu tejto možnosti.

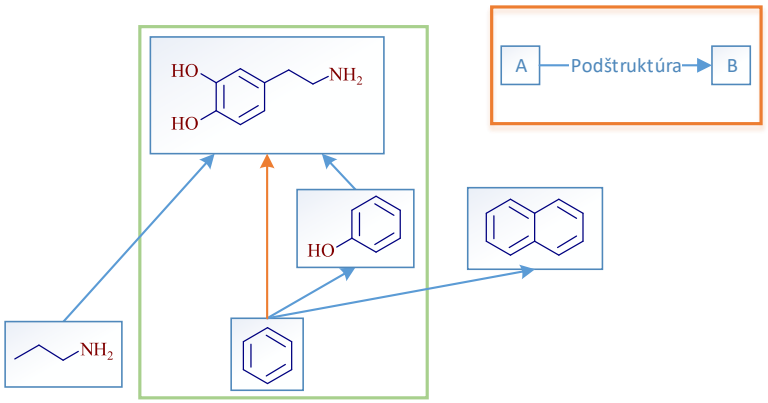
V prvom rade je potrebné zistiť, akú podporu grafového izomorfizmu majú grafové databázy, konkrétne Neo4j. V príspevku [21] sa spomína, že podporované je iba vyhľadávanie ciest, nie priamo celých podgrafov. To potvrdzuje aj implementácia triedy `PatternMatcher` v module `GraphMatching`[1], ktorá je zodpovedná za hľadanie zadaných vzorov v grafe. Tento modul bol postupne nahrádzaný implementáciou jazyka Cypher, ktorý bol popísaný v časti 3.5. Ten taktiež umožňuje v klauzule `MATCH` popisovať iba vzory, ktoré zodpovedajú cestám v grafe. Pomocou klauzule `WITH` je však možné vybrať označený vrchol na tejto ceste a začať hľadať nový vzor z tohto miesta. Vďaka tomu by bolo možné reprezentovať graf podobne ako to robia SMILES reťazce v sekcii 3.2.1.

Problémom tohto prístupu je, že nie je možné vytvoriť všeobecný Cypher dopyt, ktorý by našiel ľubovoľný podgraf v databáze na základe parametrov. Pre každý vyhľadávaný graf by teda musel byť odvodený nový dopyt, pričom jeho vytvorenie je pomerne náročné. Hlavný problém však spočíva vo veľmi neefektívnom hľadaní, kedy sa nevyhľadáva graf ako celok, ale jednotlivé cesty v ňom samostatne. Špecializované algoritmy pre vyhľadávanie podgrafov, napríklad tie predstavené v sekcii 3.4, budú v takom prípade nepochybne efektívnejšie, keďže môžu pri hľadaní využiť súvislostí, ktoré v Cypher dopyte jednoducho nie je možné popísať.

Uloženie vzťahov medzi molekulami
Ako bolo spomínané v časti 3.5, silnou stránkou grafových databáz je uchovávanie a využívanie vzťahov medzi objektmi. Vzhľadom na zameranie tejto práce by bolo zaujímavé preskúmať, akým spôsobom by mohla byť táto vlastnosť využitá pri vyhľadávaní. Jednou z možností by bolo ukladanie informácie o tom, ktorá molekula je obsiahnutá v inej. Jednalo by sa tak o zapamätanie si výsledkov podštruktúrneho vyhľadávania, vďaka čomu by výsledky pri opakovanom vyhľadávaní mohli byť vrátené takmer okamžite. V takejto databáze

[1] Repozitár: https://github.com/neo4j-contrib/neo4j-graph-matching

by vystupovali molekuly ako celok, teda jeden vrchol, a vzťahy by boli typu *podštruktúra*. Príklad takejto databázy je na obrázku 5.5. Pomocou jednoduchého dopytu v ukážke 5.1 by bolo možné získať všetky molekuly, ktoré zadaný vzor obsahujú ako podštruktúru. Predtým by sa však musela určiť molekula, ktorej daný vzor odpovedá, a jej identifikátor by sa doplnil do klauzuly `WHERE`.



Obr. 5.5: Príklad databázy s niekoľkými molekulami prepojenými vzťahom *podštruktúra*

```
1 MATCH (n)-[:SUB_STRUCT]->(m)
2 WHERE n.id = 10
3 RETURN DISTINCT m
```

Ukážka 5.1: Dopyt pre vyhľadanie vzoru v grafovej databáze

Dá sa očakávať, že počet väzieb v databáze bude veľmi rýchlo narastať a rádovo presiahne počet samotných molekúl. Aj keď Neo4j v novej verzii 3 nemá obmedzenie na počet väzieb [32], ich uloženie a spracovanie pri dopyte si stále vyžaduje výpočtové prostriedky. Možným zefektívnením by bolo využitie tranzitivity. Ak je A podštruktúra B a B podštruktúra C, prirodzene A bude podštruktúra C. Túto situáciu je možné vidieť aj na obrázku 5.5 v zelenom ráme. Vďaka tomu je oranžový vzťah nadbytočný a mohol by byť odstránený. Dopyt by sa zmenil len minimálne na `MATCH (n)-[:SUB_STRUCT*]->(m)`.

Problém v tomto prípade predstavuje možnosť parametrizácie vyhľadávania. Ak je totiž možné upraviť spôsob, akým pracuje algoritmus pri vyhľadávaní, a tým meniť výslednú množinu, v databáze nebude možné jednoducho ukladať výsledky rôzne parametrizovaných vyhľadávaní. Samozrejme by bolo možné uviesť parametre ako atribúty vzťahu, ale tým by sa databáza fragmentovala. Iným riešením by bolo vždy nájsť množinu, ktorá by obsahovala výsledok vyhľadávania pre všetky nastavenia parametrov, a do databázy pridať vzťah pre každý jej prvok. Výsledok dopytu by potom bolo nutné ešte obmedziť pomocou podštruktúrneho vyhľadávania so špecifickými parametrami. Jednalo by sa tak o akúsi formu predvýberu podobne ako je to v prípade molekulárnych odtlačkov.

Zhodnotenie predstaveného návrhu nie je jednoduché. Za určitých okolností by sa mohol skrátiť čas potrebný pre vrátenie výsledkov vyhľadávania. Záleží však o aké zrýchlenie by sa jednalo, koľko prípadov vyhľadávania by sa týmto pokrylo, koľko diskového miesta by si uloženie vzťahov vyžiadalo a tiež ako náročná by bola implementácia. Keďže sa nejedná o nevyhnutnú súčasť celkového riešenia, bolo by vhodné najskôr získať štatistiky z nasadeného systému a na ich základe sa rozhodnúť, či by bolo použitie grafových databáz v tejto situácii prínosné.

Kapitola 6

Popis implementácie

V tejto kapitole sú popísané niektoré významné implementačné detaily, na ktoré je vhodné upozorniť. Snahou však nie je podrobne popísať zdrojové kódy. Pre tento účel bola vytvorená dokumentácia a zdrojové kódy okomentované.

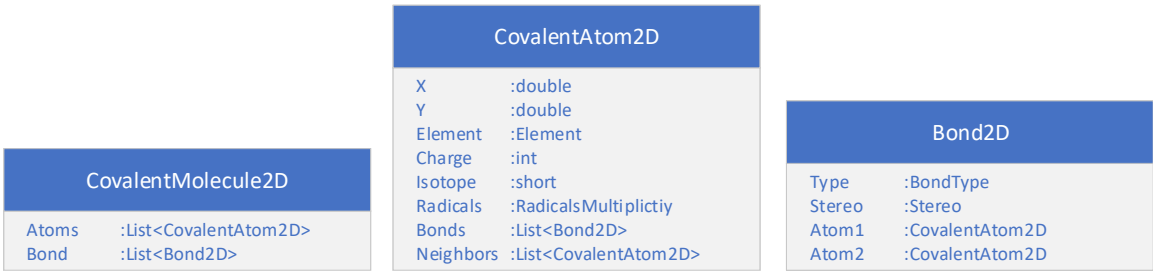
Úvodom kapitoly je v časti 6.1 v stručnosti predstavené, akým spôsobom sú v systéme chemické štruktúry reprezentované a uložené. Následne je v sekcii 6.2 popísané vytváranie InChI, InChIKey, podštruktúrnych odtlačkov a ich uloženie v databáze. Sekcia 6.3 sa zaoberá popisom implementácie algoritmov pre podštruktúrne vyhľadávanie. V sekciách 6.4 a 6.5 sú uvedené niektoré aspekty implementácií aplikačného servera a aplikácie pre webový server. Záverom sa sekcia 6.6 venuje stručnému popisu testov, ktoré overujú správnosť dôležitých operácií v rámci systému.

6.1 Reprezentácia a uloženie štruktúrnych dát

Štruktúrne dáta popisujúce chemické štruktúry je potrebné v pamäti reprezentovať spôsobom, ktorý umožní jednoduchú manipuláciu a použitie v algoritmoch naprieč systémom. Okrem toho je ale tieto dáta nutné vo vhodnom formáte ukladať na disk. Nasledujúce časti tejto sekcie poskytujú prehľad o tom, ako boli tieto súčasti systému implementované.

6.1.1 Dátová štruktúra pre reprezentáciu chemických štruktúr

Pre uloženie štruktúrnych dát existuje v projekte MolGate veľké množstvo tried, ktoré implementujú rôzne rozhrania. Tieto triedy boli prevzaté a použité pre reprezentáciu chemických štruktúr, aby sa zjednodušila následná integrácia s ďalšími systémami. Popis jednotlivých tried je možné nájsť v dokumentácii zdrojových kódov. Medzi najdôležitejšie triedy sa radí `CovalentMolecule2D`, ktorá umožňuje uložiť dáta o chemickej štruktúre definovanej v dvojrozmernom priestore. Aj keď samotný kód tejto triedy je len špecializáciou generickej triedy pomocou špecifického typu pre atómy a väzby, bude na nej objasnené, ako sú dáta uložené.



Obr. 6.1: UML diagramy tried, ktoré slúžia pre uloženie informácií o dvojrozmernej chemickej štruktúre

Obrázok 6.1 zobrazuje zjednodušené UML diagramy pre triedy `CovalentMolecule2D`, `CovalentAtom2D` a `Bond2D`, ktoré spolu ukladajú všetky potrebné informácie o štruktúre molekuly. Samotnú molekulu tvoria iba zoznamy všetkých atómov a väzieb. Každý atóm ukladá koordináty, prvok, náboj, izotop a počet radikálov. Okrem toho ukladá zoznam susediacich atómov a zoznam väzieb, ktoré vedú z tohto atómu. Každá väzba nesie informácie o svojom type, stereo orientácii a odkaz na krajné vrcholy.

6.1.2 Formát Molfile a SDF

Molfile je formát pre popis chemickej štruktúry vyvinutý firmou Molecular Design Limited (MDL), ktorá je dnes vlastníctvom firmy BIOVIA. Z historických dôvodov sa preto tento formát zvykne označovať ako MDL Molfile. Stal sa viac-menej štandardom pre výmenu štruktúrnych informácií medzi chemoinformatickými aplikáciami. Jedná sa o textový formát založený na tabuľke prepojení (angl. connection table) a jeho špecifikáciu je možné nájsť v dokumente [1]. Táto tabuľka sa skladá zo štyroch blokov. Prvý blok uvádza počet záznamov v ďalších blokoch a verziu formátu. Druhý popisuje atómy, prevažne ich pozíciu a prvok. Umožňuje popísať aj ďalšie vlastnosti, no pre tie sa v súčasnosti používa štvrtý blok. V treťom bloku sú uložené väzby popísané dvojicou indexov koncových atómov, typom väzby a stereo orientáciou. Štvrtý blok je vyhradený pre vlastnosti a umožňuje lepším spôsobom popísať napríklad náboj na atómoch alebo izotopy. Tabuľka končí riadkom `M END`.

Formát SDF vzniká združením viacerých Molfile záznamov do jedného súboru, pričom záznamy sú oddelené znakmi `$$$$`. Navyše tento formát umožňuje po skončení tabuľky prepojení uviesť dodatočné informácie o zázname pomocou asociovaných dát, ktoré sa vždy skladajú z identifikátora vo formáte `> <id>` a obsahu, ktorý je ukončený prázdnym riadkom. To umožňuje spolu so štruktúrnymi dátami uložiť dodatočné informácie, ako napríklad InChI a SMILES pre danú molekulu.

Za načítanie Molfile do grafovej štruktúry sú zodpovedné metódy z triedy `Covalent~MoleculeMolExtension`, ktoré po riadkoch čítajú vstup a na základe špecifikácie ho delia na jednotlivé polia. Trieda taktiež obsahuje metódy, ktoré umožňujú uložiť štruktúrne dáta v tomto formáte. Okrem toho je pripravená trieda `SdfReader`, ktorá umožňuje postupne čítať jednotlivé záznamy a ich asociované dáta z formátu SDF, a trieda `SdfWriter` umožňuje do tohto formátu štruktúrne dáta molekúl zapisovať.

6.1.3 Serializovaný formát MPM

Formát Molfile je určený predovšetkým pre prenos medzi rôznymi systémami. Nie je ale vhodný, pokiaľ je potrebné dáta rýchlo načítať alebo uložiť čo najúspornejšie, keďže sa

jedná o textový formát. Druhý z týchto problémov je možné čiastočne riešiť komprimáciou, napr. metódou DEFLATE, no závažnejší je prvý problém, keďže dáta bude potrebné počas vyhľadávania čo najrýchlejšie pripraviť na použitie. Z tohto dôvodu bolo rozhodnuté vytvoriť vlastný formát založený na binárnej serializácii.

	Formát			
	MPM 1	MPM 2	MPM 3	MPM 4
Veľkosť (B)	553.167	540.187	400.58	519.614
Deserializácia (µs)	12.73	10.411	8.63	7.91

Tabuľka 6.1: Porovnanie viacerých spôsobov serializácie údajov o atómoch v chemických štruktúrach

V prvom rade bola zvolená knižnica MessagePack for C#[1]. Na jej stránkach sa uvádza, že sa jedná o najrýchlejšiu serializačnú knižnicu pre C#. Veľkou výhodou je tiež vstavaná podpora pre LZ4 kompresiu. Následne bolo potrebné rozhodnúť, akým spôsobom štruktúrne dáta serializovať. Informácie pre každý atóm mohli byť uložené v oddelených objektoch, ktoré by sa serializovali každý zvlášť alebo v niekoľkých poliach spoločných pre celú molekulu, napr. pole prvkov, pole izotopov, atď. Aby sa zistilo, ktorá forma reprezentácie je najvhodnejšia, boli vytvorené štyri formáty `MessagePackMolecule` (MPM) 1 až 4, ktoré ukladajú údaje o atómoch rôznou formou. Trieda `CovalentMoleculeMpmExtension` obsahuje všetky tieto formáty a implementuje metódy pre ich serializáciu a deserializáciu. Následne bolo vykonané meranie na 100000 náhodných molekulách z databázy PubChem, ktoré malo za cieľ určiť priemernú veľkosť serializovaných dát a dobu potrebnú na spätnú deserializáciu. Meranie času bolo desaťkrát zopakované. Spriemerované výsledky je možné nájsť v tabuľke 6.1. Ako najoptimálnejšia verzia formátu sa javí MPM 3, kedy sú súradnice a prvky uložené v poliach spoločných pre celú molekulu a menej bežné vlastnosti atómov, ako napr. izotopy, sú uložené vo forme riedkych polí, kde každý prvok obsahuje index atómu a hodnotu danej vlastnosti.

	Formát			
	Molfile	Molfile (DEFLATE)	MPM (`double`)	MPM (`int`)
Veľkosť (B)	4431.555	636.61	614.941	465
Deserializácia (µs)	Nemerané	259.283	Nemerané	64.91

Tabuľka 6.2: Porovnanie formátov MPM a SDF. Verzie, u ktorých nebol meraný čas deserializácie, sú uvedené iba pre porovnanie veľkosti, ich použitie by vzhľadom na existenciu lepšej varianty nedávalo zmysel.

Po pridaní informácií o väzbách bolo možné formát MPM porovnať s formátom Molfile komprimovaným pomocou metódy DEFLATE. Aby bolo porovnanie spravodlivejšie, bol dátový typ pre súradnice vo formáte MPM zmenený z `double` na `int`, keďže Molfile podporuje pre súradnice iba hodnoty s pevnou desatinnou čiarkou, ktorých rozsah je možné reprezentovať dátovým typom `int`. Porovnanie je možné nájsť v tabuľke 6.2. Meranie bolo vykonané na 20000 náhodných molekulách a doba deserializácie je výsledkom spriemerova-

[1]Repozitár: https://github.com/neuecc/MessagePack-CSharp

nia desiatich opakovaní. Výsledný formát MPM má oproti komprimovanému Molfile o 27 % menšiu veľkosť a umožňuje až štvornásobne rýchlejšiu deserializáciu do dátovej štruktúry molekuly.

6.1.4 Uloženie v súborovom systéme NTFS

Uloženie štruktúrnych dát ako súbory v súborovom systéme formou stromovej štruktúry, ktorá bola zvolená na základe meraní v časti 7.2, implementuje trieda `DiskMoleculeStorage`. Parametrom konštruktora je cesta k zložke, v ktorej sa úložisko nachádza, prípadne sa má vytvoriť. V tejto zložke sa vytvorí konfiguračný súbor `molstore.conf` vo formáte JSON, ktorý špecifikuje predovšetkým počet súborov v každej zložke a počet úrovní stromu. Súčasťou je taktiež počítadlo uložených záznamov, aby bolo možné po reštarte aplikácie pokračovať v plnení úložiska. Keďže operácia odstránenia dát bude veľmi ojedinelá, opakované použitie identifikátora uvoľneného zmazaním záznamu nie je implementované. Pre každý vložený záznam sa použije aktuálna hodnota počítadla ako identifikátor a počítadlo sa následne inkrementuje. Podľa identifikátora je potom záznam uložený do príslušnej cesty, ktorá vznikne postupným delením a použitím operácie modulo. Záznam s identifikátorom 98376 by bol v úložisku s tromi úrovňami a sto súbormi v zložke uložený do súboru `storage/9/83/76.mpm`. Pre každú molekulu sú štruktúrne dáta v serializovanom formáte MPM uložené do samostatného súboru. Aby sa zabránilo modifikácii úložiska z viacerých procesov súčasne, je konfiguračný súbor otvorený po celú dobu behu aplikácie s argumentom `FileShare.None`, ktorý spôsobí výnimku v druhom procese, ak by sa snažil tento súbor otvoriť.

6.1.5 Uloženie v databáze MongoDB

Implementáciu databázového úložiska štruktúrnych dát pre databázu MongoDB obsahuje trieda `DatabaseMoleculeStorage`. Komunikáciu s databázou zabezpečuje oficiálny MongoDB ovládač pre C#[1]. Pri spustení aplikácie sa vytvorí spojenie s databázovým serverom a skontroluje sa existencia databázy. Ak nie je nájdená, dôjde k jej založeniu a vytvoria sa dve kolekcie. V kolekcii `molecules` sú uložené samotné štruktúrne dáta a kolekcia `sequences` slúži pre uloženie počítadiel určených predovšetkým pre generovanie identifikátorov záznamov.

```
1 var findTask = _moleculeCollection.FindAsync(new
      FilterDefinitionBuilder<MoleculeEntry>().In(x => x.MolId, IDs));
2 var cursor = await findTask;
3 var entries = await cursor.ToListAsync();
```

Ukážka 6.1: Kód pre asynchrónne získanie záznamov z databázy MongoDB

Najdôležitejšou operáciou je asynchrónne získanie požadovaných záznamov z databázy. To zabezpečuje metóda `RetrieveManyPackedAsync`, ktorej najvýznamnejšiu časť je možné vidieť v ukážke 6.1. Na riadku 1 dochádza k vytvoreniu dopytu, ktorý má za úlohu vrátiť všetky záznamy, ktorých hodnota `MolId` sa nachádza v zozname identifikátorov požadovaných záznamov `IDs`. Následne sa na riadku 2 asynchrónne čaká na vykonanie dopytu. Po jeho dokončení sú z databázy na riadku 3 opäť asynchrónne získané záznamy, ktoré sú výsledkom dopytu. V prípade paralelného spracovávania záznamov je možné vďaka asynchrón-

[1] Odkaz na stiahnutie: https://docs.mongodb.com/ecosystem/drivers/csharp/

nym čakaniam na výsledok dopytov pokračovať v spracovávaní dát vo zvyšných vláknach, pretože čakajúce vlákna neobsadzujú procesor.

6.2 Vytvorenie a uloženie štruktúrnych metadát

Popisu ukladaných štruktúrnych metadát sa venovala sekcia 5.4.2. Tieto metadáta sú v PostgreSQL databáze v súčasnosti uložené v rámci jedinej tabuľky, kde jedna molekula predstavuje jeden záznam. Najdôležitejšími údajmi sú tri bloky InChIKey a podštruktúrny odtlačok, ktoré umožňujú rýchle identické a podštruktúrne vyhľadávanie. Táto sekcia popisuje spôsob ich vytvárania, ako aj ich následné použitie v databáze pri vyhľadávaní.

6.2.1 Generovanie InChI a InChIKey

Pri návrhu systému bolo v časti 5.3.1 spomenuté, že pre generovanie InChI a InChIkey sa použije referenčná knižnica implementovaná v jazyku C. V prostredí .Net sa odlišujú manažované a natívne knižnice. Referenčnú knižnicu je možné považovať za natívnu, pretože nepoužíva platformu CLR a jej kód je skompilovaný priamo na inštrukcie pre procesor určitej architektúry. Manažované knižnice využívajú platformu CLR, ktorá zabezpečuje napríklad automatické uvoľňovanie pamäti alebo kompiláciu za behu (angl. just-in-time compilation, JIT). Práve JIT umožňuje, že knižnica môže byť skompilovaná do medzikódu, pre ktorý sa vytvoria príslušné inštrukcie až pri spustení na konkrétnom počítači. Vďaka tomu je možné mať pre všetky architektúry jednu verziu dynamickej knižnice. Problémy však nastávajú, ak je potrebné použiť natívnu knižnicu z manažovanej aplikácie využívajúcej prostredie .Net. Prvý problém súvisí s volaním funkcií a vytváraním dátových štruktúr z natívnej knižnice. Tým druhým je načítanie natívnej knižnice so správnou architektúrou za behu.

Volanie funkcií z natívnej knižnice je možné riešiť pomocou metódy `PInvoke`, ktorá však vyžaduje uvedenie názvu knižnice a špecifikovanie konvencie pre volanie funkcií priamo v kóde. V konečnom dôsledku je toto riešenie neelegantné a náročné na použitie v prípade, že je potrebné využiť natívne dátové štruktúry. V tejto práci bol preto zvolený prístup, kedy sa vytvorí manažovaná knižnica v jazyku C++\CLI, ktorá zaobaľuje funkcionalitu natívnej knižnice. Takáto knižnica umožňuje jednak linkovať natívnu knižnicu, vďaka čomu je možné priamo používať jej funkcie a dátové štruktúry, a tiež vytvorenie manažovaných metód, ktoré slúžia ako rozhranie knižnice a je možné ich volať v jazyku C# bežným spôsobom.

K vytvoreniu takejto knižnice pre generovanie InChI a InChIKey slúži kód v projekte `InchiWrapper`. Knižnica obsahuje manažovanú triedu `InchiGenerator`, ktorej metódy je možné používať v jazyku C#. Tieto metódy umožňujú napríklad získanie InChI priamo na základe dátovej štruktúry molekuly alebo prevod InChI reťazca na InChIKey. Metódy najskôr prevedú daný vstup na dátové štruktúry natívnej knižnice, zavolajú zodpovedajúcu funkciu, a jej výsledok spätne transformujú na manažovanú dátovú štruktúru.

Druhý problém s načítaním správnej natívnej knižnice pre konkrétnu architektúru sa podarilo vyriešiť vďaka mechanizmu načítania dynamických knižníc až pri prvom použití. To znamená, že pri štarte aplikácie sa načítajú iba základné knižnice a zvyšné dynamické knižnice sú procesom aplikácie načítané až v prípade, že sa použije nejaká metóda alebo trieda knižnice. To umožňuje dynamicky za behu aplikácie zvoliť správne knižnice. Pre natívne knižnice sú vytvorené zložky označené názvom architektúry, napr. `x86` a `x64`, do ktorých sú vložené príslušné dynamické knižnice, a obe zložky sú skopírované do zložky obsahujúcej spustiteľný súbor aplikácie. Následne sa po spustení určí architektúra, napr. porovnaním

veľkosti dátového typu pre ukazovatele, ktorá je pre 32-bitovú architektúru 4 B a pre 64-bitovú architektúru 8 B. Na základe tejto hodnoty sa pridá do premennej Path viditeľnej iba pre konkrétny proces cesta, ktorá zodpovedá zložke s natívnou knižnicou správnej architektúry, čiže pre 64-bitovú architektúru sa pridá zložka `x64`. To sa deje pomocou metódy `Environment.SetEnvironmentVariable` s prvým parametrom `"Path"`. V prípade, že proces potrebuje načítať natívnu knižnicu, skontroluje okrem iného svoju premennú Path, kde sa teraz už nachádza cesta k tejto knižnici, a preto načítanie prebehne úspešne. Okrem natívnej knižnice je potrebné načítať aj správnu manažovanú knižnicu, ktorá ju zaobaľuje. Pri tejto knižnici nie je možné využiť JIT, pretože priamo linkuje natívnu knižnicu konkrétnej architektúry. K pridaniu cesty k zložke obsahujúcej správnu manažovanú knižnicu je možné použiť metódu `AppDomain.CurrentDomain.AppendPrivatePath`.

Toto riešenie je možné nájsť implementované v projekte `NativeLibLoader`, kde trieda `PathLibLoader` obsahuje metódu `RegisterLibs`. Túto metódu je nutné zavolať bezprostredne po štarte aplikácie, vďaka čomu sa zaregistrujú správne cesty ku knižniciam a proces ich tak bude schopný v prípade potreby úspešne načítať.

6.2.2 Uloženie a indexovanie InChIKey

V databáze nie sú dáta jednotlivých InChIKey blokov uložené v pôvodnej textovej Base26 reprezentácii, ale komprimované na polia bajtov. Táto komprimácia je jednoduchá a spočíva v uložení každého znaku iba na 5 bitov. Oproti pôvodnému bajtu pre každý znak sa v prípade prvých dvoch blokov dosiahne kompresie o 37.5 %, vďaka čomu bude veľkosť dát a indexu v databáze menšia, a taktiež bude vyhľadávanie vzhľadom na menšiu dĺžku polí rýchlejšie. Tretí blok je ponechaný v pôvodnej forme, keďže sa jedná iba o jeden znak.

```
1  explain analyze Select "Id" from mol."Molecules" where "InchiKeyBlock1" =
       E'\\x0F1C2913CB69D50005' and "InchiKeyBlock2" = E'\\x760E44ED4A' and
       "InchiKeyBlock3" = 78;
2
3                                       QUERY PLAN
4  ------------------------------------------------------------------------------------
5  Index Scan using "Molecules_IX_InchiKey" on "Molecules" (cost=0.44..8.46 rows=1
       width=4) (actual time=0.267..0.268 rows=1 loops=1)
6  Index Cond: (("InchiKeyBlock1" = '\x0f1c2913cb69d50005'::bytea) AND ("InchiKeyBlock2"
       = '\x760e44ed4a'::bytea))
7  Filter: ("InchiKeyBlock3" = 78)
8  Planning time: 0.444 ms
9  Execution time: 0.286 ms
10 (5 rows)
```

Ukážka 6.2: Analýza dopytu pre vyhľadanie záznamu podľa InChIKey

Pre prvé dva bloky bol v databáze vytvorený index, ktorý umožňuje veľmi rýchle nájdenie konkrétneho záznamu podľa InChIKey. Bez indexu by bolo nutné porovnať všetky záznamy v databáze a doba vyhľadania záznamu by tak s veľkosťou databázy lineárne rástla. Efekt indexovania je možné vidieť v ukážke 6.2. Pre prvé dva bloky databázový plánovač skutočne využíva index a získané záznamy sú ďalej redukované pomocou filtra, ktorý porovnáva hodnotu tretieho bloku. Pre viac ako 9 miliónov záznamov trvalo vyhľadanie konkrétneho záznamu iba 0.286 ms, z čoho doba vyhľadávania v indexe bola približne 0.267 ms.

6.2.3 Odtlačky PatternFingerprint z knižnice RDKit

Pre tvorbu podštruktúrnych odtlačkov bola použitá knižnica RDKit. Ako bolo spomenuté v časti 5.3.2, jediným vhodným odtlačkom pre podštruktúrne vyhľadávanie sa ukázal byť PatternFingerprint. Aby mohol byť tento odtlačok vygenerovaný, je najskôr potrebné vytvoriť dátovú štruktúru popisujúcu molekulu z knižnice RDKit, konkrétne inštanciu triedy `ROMol`. Prevod na túto štruktúru sa však ukázal problematický hlavne kvôli chýbajúcej dokumentácii ohľadom správneho spôsobu určenia stereochémie. Keďže proces vytvárania odtlačkov nie je časovo kritický a vo veľkej miere sa táto operácia týka inicializácie databázy, bol nakoniec zvolený prevod cez Molfile formát, do ktorého sú informácie o molekule exportované a knižnicou RDKit správne načítané do potrebnej dátovej štruktúry. Za samotné vytvorenie odtlačku je zodpovedná metóda `RDKFuncs.PatternFingerprintMol`, ktorej výstup je objekt triedy `ExplicitBitVector`. Ten je potrebné konvertovať na pole bajtov, ktoré sa vo zvyšku aplikácie používa pre reprezentáciu odtlačku. Tým je proces vytvorenia odtlačku ukončený.

SMARTS reťazec	Význam
`[*]~[*]`	Cesta dĺžky 2
`[*]~[*]~[*]`	Cesta dĺžky 3
`[R]~1~[R]~[R]~1`	Kružnica s 3 vrcholmi
`[*]~[*](~[*])~[*]`	Cesta dĺžky 3 s vetvením na druhom vrchole
`[R]~1[R]~[R]~[R]~1`	Kružnica so 4 vrcholmi
`[*]~[*]~[*](~[*])~[*]`	Cesta dĺžky 4 s vetvením na treťom vrchole
`[R]~1~[R]~[R]~[R]~[R]~1`	Kružnica s 5 vrcholmi
`[R]~1~[R]~[R]~[R]~[R]~[R]~1`	Kružnica so 6 vrcholmi
`[R](@[R])(@[R])~[R]~[R](@[R])(@[R])`	Spojnica medzi dvoma cyklami.
`[R](@[R])(@[R])~[R]@[R]~[R](@[R])(@[R])`	Spojnica medzi dvoma cyklami, ktorá prechádza tretím cyklom
`[*]~[R](@[R])@[R](@[R])~[*]`	Časť kružnice s vetvením na druhom a treťom vrchole
`[*]~[R](@[R])@[R]@[R](@[R])~[*]`	Časť kružnice s vetvením na druhom a štvrtom vrchole
`[*]`	Vzor značiaci samostatné vrcholy

Tabuľka 6.3: Význam vzorov použitých v odtlačku PatternFingerprint

Je však vhodné vedieť, ako je tento odtlačok implementovaný priamo v knižnici RDKit. Zjednodušene je tento proces možné popísať ako vyhľadávanie výskytu určitej množiny vzorov a zakódovanie ich výskytov do odtlačku. Na počiatku je vytvorený nulový bitový vektor požadovanej dĺžky. Tabuľka 6.3 uvádza SMARTS vzory, ktoré sa vo vstupnej molekule hľadajú pomocou podštruktúrneho algoritmu VF2. Po nájdení výskytov jedného vzoru je inicializované jedno 32-bitové číslo hodnotou, ktorá vznikne sčítaním indexu SMARTS vzoru a počtu atómov a väzieb molekuly. K tomuto číslu sa pre každý výskyt v cykle prihašuje hodnota `0xBEEF` a výsledok slúži ako index bitu, ktorého hodnota sa zmení na 1. Týmto spôsobom sa kóduje do odtlačku počet výskytov daného vzoru. Okrem toho sa pre každý výskyt zmení hodnota ešte jedného bitu, ktorého index sa vypočíta zahašovaním indexu

vzoru s niektorými vlastnosťami atómov a väzieb, ktoré tvoria podgraf. Tieto vlastnosti sú protónové číslo atómu a typ väzby. K hašovaniu hodnôt sa vo veľkej miere používa funkcia `hash_combine` z knižnice Boost.

6.2.4 Zásuvný modul pre PostgreSQL

Pre databázu, v ktorej sú uložené podštruktúrne odtlačky, bolo potrebné vytvoriť kompilovaný zásuvný modul implementujúci vstavanú funkciu pre porovnanie odtlačkov. Tento zásuvný modul implementuje projekt `MolSearchDbExtension`. Princíp vstavanej funkcie `fingerprint_substruct_test` je pomerne jednoduchý. Po získaní dát oboch odtlačkov, ktoré sú predané ako parametre, začne oba odtlačky prechádzať po bajtoch. Ak odtlačok vzoru nazveme `fp1` a odtlačok záznamu `fp2`, musí pre každý bajt s indexom `i` platiť `(fp1[i] & fp2[i]) == fp1[i]`, kde `&` je v bežnom význame bitového prieniku. V prípade, že všetky bajty spĺňajú túto podmienku, funkcia vráti hodnotu `true`, inak `false`.

Implementovaním zásuvných modulov pre PostgreSQL sa zaoberá online dokument [27]. K implementácii požadovanej funkcie je nutné vedieť prevziať argumenty, čiže dva odtlačky, zistiť ich dĺžku a získať ich dáta, teda polia bajtov. K získaniu argumentov sa používa funkcia `PG_GETARG_BYTEA_PP`. Napríklad `PG_GETARG_BYTEA_PP(0)` získa prvý argument. Ten bude v prípade implementovanej funkcie typu `bytea`. K získaniu dĺžky slúži funkcia `VARSIZE_ANY_EXHDR` a návratová hodnota funkcie `VARDATA_ANY` je ukazovateľ na dáta, v tomto prípade `uint8_t*`. K vráteniu pravdivostnej hodnoty zo vstavanej funkcie je potom používaná funkcia `PG_RETURN_BOOL`.

6.3 Algoritmy pre podštruktúrne vyhľadávanie

Všetky algoritmy pre podštruktúrne vyhľadávanie, čiže hľadanie izomorfného podgrafu, boli prepísané do jazyka C# a upravené jednak k dosiahnutiu požadovaného výsledku, ale aj za účelom zrýchlenia. Tieto úpravy budú pre každý algoritmus popísané zvlášť v nasledujúcich častiach. Ich implementácie je možné nájsť v projekte `SearchAlgorithms` v zložkách pomenovaných podľa algoritmov. Tento úvodný text sa zaoberá aspektami spoločnými pre všetky algoritmy. V prvom rade je to spoločná abstraktná trieda `SubgraphIsomorphismSolver`, ktorá abstrahuje konkrétny spôsob vyhľadávania a deklaruje metódy pre nájdenie výskytu vzoru v grafe. Triedy jednotlivých algoritmov sú vytvorené dedením z tejto základnej triedy. Jedná sa o generickú triedu, pre ktorú je možné určiť typ vrcholu `TVertex`, hrany `TEdge` a triedy implementujúcej operácie porovnania vrcholov a hrán `TComparator`. Jedná sa tak o všeobecnú implementáciu, ktorá nie je priamo naviazaná na chemické štruktúry. K tomu dochádza až špecializovaním uvedených typov. Práve rôzne implementácie typu `TComparator` umožňujú meniť, akým spôsobom dochádza k porovnaniu grafov. Napr. trieda `MoleculeComparator` implementuje toto porovnanie tak, že pre vrcholy sa musí zhodovať prvok atómu, a ak má vrchol vzoru nejakú ďalšiu vlastnosť, napríklad určený náboj, musí sa tento náboj na priradených vrcholoch zhodovať. Hrany sú porovnávané iba na základe typu väzby.

Ďalej bolo nutné vytvoriť jednotné rozhranie pre prácu s grafmi, ktoré by mohli jednotlivé algoritmy využívať. Toto rozhranie bolo nazvané `IGraph` a umožňuje zistiť napr. počet vrcholov grafu, stupeň určitého vrcholu, nájsť hranu medzi dvoma vrcholmi alebo získať konkrétneho suseda vrcholu. Rozhranie potom implementuje trieda `MoleculeGraphAdapter`, ktorá prevádza triedy uvedené v časti 6.1.1 na toto rozhranie. Táto abstrakcia opäť umožňuje, aby algoritmy mohli byť použité pre ľubovoľné grafy.

6.3.1 Algoritmus VF2

Implementácia tohto algoritmu vychádza z kódu knižnice VFLib prepísaného do jazyka C#. Táto knižnica umožňuje hľadanie grafového izomorfizmu, a taktiež indukovaného a neindukovaného podgrafového izomorfizmu. Telo algoritmu je pre všetky prípady rovnaké, mení sa len implementácia operácií, ktoré vyhodnocujú aktuálne priradenie vrcholov. Konkrétne neindukovaný izomorfizmus implementuje v knižnici VFLib trieda `VF2MonoState`. Po prepísaní kódu bolo nutné pôvodnú implementáciu upraviť tak, aby nezohľadňovala orientáciu hrán. Toho sa docielilo zlúčením cyklov v metódach `IsFeasiblePair`, `AddPair` a `BackTrack`, ktoré v pôvodnej implementácii iterovali zvlášť cez vstupné a výstupné hrany. Miesta, kde k tomuto zlúčeniu došlo, sú v kóde triedy `UnorientedMonoState` označené komentárom.

Z hľadiska optimalizácie boli vykonané dve zmeny, ktoré výrazne zrýchlili beh algoritmu oproti pôvodnej implementácii. Obe zmeny sú v metóde `IsFeasiblePair`, ktorá kontroluje, či stav môže viesť k riešeniu. Prvou je kontrola stupňa vrcholov. Ak má totiž vrchol vo vzorovom grafe väčší počet susedov než priradený vrchol z cieľového grafu, tento stav určite nepovedie k riešeniu. Druhou bolo optimalizovanie porovnania hrán. V pôvodnej implementácii sa najskôr zisťuje, či existuje hrana medzi dvoma vrcholmi a ak áno, získajú sa podľa krajných vrcholov jej vlastnosti, ktoré sa porovnajú s hranou vzorového grafu. Problémom je, že tak dochádza k dvojnásobnému hľadaniu hrany a operácia porovnania hrán je v algoritme veľmi často volaná. Implementácia bola zmenená tak, aby pri nájdení hrany bol vrátený jej index, v opačnom prípade je vrátená hodnota -1. Vďaka tomu už nie je nutné hranu znova hľadať pri získavaní jej vlastností, ale získa sa pomocou indexu. Tieto pomerne jednoduché zmeny viedli k priemernému skráteniu doby hľadania podgrafu o 27 %. V porovnaní s implementáciou VF2 v knižnici RDKit je potom táto implementácia priemerne o 3 % rýchlejšia. Meranie bolo vykonané vyhľadávaním vzorov v databáze tak, ako je to popísané v časti 7.1, kde vzhľadom na minimálny rozdiel v dobe behu medzi implementáciami nie je tá z knižnice RDKit uvedená.

6.3.2 Algoritmus Glasgow

Implementácia algoritmu Glasgow tiež vychádza z referenčnej knižnice dostupnej v online repozitári[1]. Bola použitá implementácia bez paralelizmu na úrovni hľadania izomorfizmu, keďže efektívnejšie bude spracovávať viacero grafov súčasne. Opäť však došlo k niekoľkým úpravám. Z algoritmu bola vyňatá podpora pre slučky v grafoch, ktoré sa v chemických štruktúrach nevyskytujú. Taktiež neboli využité doplnkové grafy, pretože ich zostrojenie trvalo v prípade chemických štruktúr dlhšie ako samotné hľadanie izomorfizmu bez nich. Algoritmus v pôvodnej implementácii neuvažuje vlastnosti vrcholov a hrán, preto bolo nutné doplniť ich kontrolu. Porovnanie vlastností vrcholov sa deje pri inicializácii domén, zatiaľ čo porovnanie hrán je možné až pri behu algoritmu v metóde `Assign`, ktorá overuje platnosť aktuálneho priradenia vrcholov.

Profilovaním a optimalizáciami kódu sa podarilo základný prepis algoritmu zrýchliť až päťkrát. Výrazne optimalizovaná bola napr. metóda pre inicializáciu domén, ktorá je zodpovedná za určenie možných priradení vrcholov vzorového grafu na vrcholy cieľového grafu. Význam domén bol vysvetlený v časti 3.4.3. Proces ich vytvorenia je iteratívny, pretože zmena jednej domény môže viesť k zmene ďalších. V pôvodnej implementácii sa však pri každej iterácii hľadali nanovo všetky možné priradenia. Úvahou je však možné dospieť

[1] Repozitár: https://github.com/ciaranm/cp2015-subgraph-isomorphism/

k tomu, že domény sa môžu iba zmenšovať, teda z prvotnej množiny možných priradení sa budú prvky odstraňovať, ale nikdy nemôžu do množiny pribudnúť. Pri prvej iterácii sa preto volá metóda `InitialUpdateDomains` a pri nasledujúcich už optimalizovaná metóda `SuccessiveUpdateDomains`. Optimalizácia bola vykonaná na základe meraní, ktoré ukazovali, že podstatnú časť doby behu algoritmu tvorí práve inicializácia domén, nie samotné hľadanie izomorfizmu. Vďaka tejto zmene sa podarilo inicializáciu domén skrátiť o 76 %, čo viedlo k priemernému skráteniu celkovej doby behu algoritmu o 63 %. Taktiež boli optimalizované dátové štruktúry, napríklad použitie poľa pravdivostných hodnôt `bool[]` namiesto pôvodnej triedy `BitSet` v kritických miestach, čo umožňuje rýchlejší prístup k hodnote na úkor pamäťovej náročnosti.

6.3.3 Algoritmus RI

Implementácia algoritmu RI nie je priamo verejne dostupná a bola získaná kontaktovaním autorov článku [2]. Opäť sa jednalo o implementáciu v jazyku C++, preto prvým krokom bolo prepísanie do jazyka C#. Následne bolo potrebné vyriešiť získanie výsledku, ktorý bol v pôvodnej implementácii iba vypísaný do súboru. Algoritmus navyše pracuje tak, že nájde všetky výskyty vzoru v grafe, nie iba jeden. Aby sa táto vlastnosť zachovala, bolo zvolené navrátenie hodnoty pomocou kľúčového slova `yield`. To umožňuje vytvoriť takzvaný generátor, ktorého návratovou hodnotou je objekt s rozhraním `IEnumerable` umožňujúci postupnú iteráciu cez výsledky pomocou metódy `MoveNext` alebo cyklu `foreach`. Je tak možné efektívne získať iba jeden výskyt alebo v prípade potreby nájsť aj tie zvyšné. Podobne ako pri algoritme VF2, aj RI rozlišoval orientáciu hrán, k čomu dochádzalo v metóde `Solve` v triede `Solver`. Pôvodne sa rozlišovalo, či sa do vrcholu prislúchajúcemu súčasnému stavu prišlo vstupnou alebo výstupnou hranou. Zlúčením hodnôt `in` a `out` typu predka v radení premenných `MaMaParentType` a upravením podmienky v spomínanej metóde sa podarilo vytvoriť implementáciu, ktorá nerozlišuje orientáciu hrán. Algoritmus bol pomerne dobre optimalizovaný a ani jeho profilovaním sa nepodarilo vytvoriť úpravy, ktoré by viedli k významnému zrýchleniu. V tomto ohľade sa tak jedná o takmer pôvodnú implementáciu.

6.4 Aplikačný server

Aplikačný server je implementovaný ako konzolová aplikácia v projekte `ApplicationServer`. Hlavnú časť tvorí trieda správcu úloh `JobManager`, ktorá implementuje metódy vykonávajúce identické a podštruktúrne vyhľadávanie. Pri oboch úlohách sa najskôr prevedie vstup v podobe textového reťazca vo formáte Molfile na grafovú štruktúru popísanú v časti 6.1.1. V prípade identického vyhľadávania sa pokračuje vytvorením InChI a InChIKey, podľa ktorých je vyhľadaný záznam v databáze. Podštruktúrne vyhľadávanie je komplikovanejšie. Pre vstup sa vytvorí podštruktúrny odtlačok, ktorý sa použije pre získanie množiny kandidátov z databázy. Ďalej je nutné pre každého kandidáta skontrolovať skutočný výskyt vzoru podštruktúrnym algoritmom. K tomu je potrebné načítať štruktúrne dáta pre danú molekulu a nájsť výskyt vzoru pomocou algoritmu pre hľadanie izomorfného podgrafu. Na základe meraní v sekcii 7.1 bol zvolený algoritmus RI. Túto úlohu je možné jednoducho paralelizovať a kontrolovať výskyt vzoru pre viacero kandidátov súčasne. Spracovanie každého kandidáta samostatne by však vzhľadom na réžiu spojenú s vytváraním vlákien a prepínaním kontextov bolo neefektívne. Množina kandidátov je preto rozdelená na tzv. dávky. Každú dávku tvorí 1000 kandidátov. Jedna dávka je spracovaná sekvenčne a k paralelizmu dochádza na úrovni dávok. To okrem iného umožňuje aj načítanie štruktúr-

nych dát pre celú dávku jedným dopytom. Aby mohli byť výsledky pridávané do množiny výsledkov z viacerých vlákien súčasne, bola použitá kolekcia `ConcurrentBag`. Podrobnejšie je spôsob paralelizácie popísaný v nasledujúcej časti 6.4.1.

Aplikácia po spustení zaregistruje obslužné metódy pre požiadavky prijímané cez RabbitMQ, ktoré využívajú inštanciu správcu úloh pre spustenie nových vyhľadávaní alebo získanie výsledkov. K tomu je nutné najskôr vytvoriť objekt `IBus` z knižnice EasyNetQ, ktorý predstavuje spojenie s RabbitMQ serverom. Pre tento účel slúži metóda `CreateBus`, ktorej sú ako textový reťazec predané parametre spojenia, predovšetkým adresa servera. K registrácii obslužných metód potom dochádza volaním metódy `Respond` na objekt `IBus`. Knižnica EasyNetQ vo východzom nastavení pomenuje fronty podľa typu prijímanej a navrátenej správy. Toto chovanie je možné zmeniť volaním metódy `WithQueueName` na konfiguračný objekt, ktorý je voliteľný parameter metódy `Respond`. Toho sa využíva pre vytvorenie samostatných front pre každý aplikačný server, ktoré slúžia pre spätné dopytovanie na priebeh a výsledok vyhľadávania.

6.4.1 Paralelizácia podštruktúrneho vyhľadávania

K implementácii paralelizmu bola pôvodne použitá funkcia `Parallel.ForEach`, ktorá je dostupná v základnom prostredí .Net. Tá však nepodporuje asynchrónne operácie, ktoré sú použité napr. pri získaní štruktúrnych dát z databázy. Preto bola funkcia nahradená implementáciou `ParallelForEachAsync` z knižnice AsyncEnumerable[1], ktorá tento problém rieši. Táto funkcia však neumožňuje okrem nastavenia miery paralelizmu žiadne ďalšie voľby. Ďalším problémom bolo, že k načítaniu štruktúrnych dát a overeniu výskytu vzoru dochádzalo v rámci spoločného vlákna. Každé vlákno teda asynchrónne čakalo na získanie dát a až následne hľadalo výskyt pomocou algoritmu. To malo za následok neefektívne využitie paralelizmu. Zvýšenie miery paralelizmu neviedlo k lepším výsledkom, pretože sa tým vytvorilo mnoho súčasných požiadaviek na databázový server, čo spôsobilo degradáciu jeho výkonu.

Lepšie riešenie, ktoré viedlo k efektívnejšiemu využitiu prostriedkov a až dvojnásobnému zrýchleniu, využíva pôvodne neznámu knižnicu TPL Dataflow, ktorá je taktiež súčasťou prostredia .Net. Princíp tejto knižnice je výrazne odlišný od bežných spôsobov paralelizácie. Knižnica umožňuje vytvoriť tzv. bloky a tie následne prepojiť, čím vznikne zreťazená linka (angl. pipeline). Existujú rôzne typy blokov, ktoré sa líšia spôsobom, akým pracujú so vstupmi a výstupmi. K implementácii paralelného spracovania kandidátov boli potrebné dva typy blokov, konkrétne `TransformBlock` a `ActionBlock`. Prvý typ umožňuje každú vstupnú hodnotu transformovať na novú výstupnú hodnotu a tú predať ďalej do reťazca. Pomocou tohto bloku bolo implementované načítanie štruktúrnych dát. Každá dávka zložená z identifikátorov požadovaných záznamov sa v rámci tohto bloku transformuje na zoznam serializovaných štruktúrnych dát. Následne je použitý `ActionBlock`, ktorý tento zoznam preberá na svojom vstupe. Telo tohto bloku je tvorené cyklom, v ktorom sa sekvenčne pre každý prvok zoznamu zostrojí dátová štruktúra molekuly a skontroluje výskyt vzoru. V prípade nájdenia podgrafu je identifikátor pridaný do množiny výsledkov. Výhodou tohto prístupu je, že načítanie štruktúrnych dát sa teraz deje oddelene od behu algoritmu. Vďaka tomu môže načítanie dát z databázy prebiehať bez prerušenia maximálnou rýchlosťou a v prípade, že by druhý blok nestíhal tieto dáta spracovávať, budú vkladané na vstupnú frontu bloku. Taktiež je možné oba bloky parametrizovať samostatne, čiže napríklad zvlášť určiť mieru paralelizmu pre získavanie dát z databázy a beh algoritmov.

[1] Repozitár: https://github.com/tyrotoxin/AsyncEnumerable

6.5 Webový server

Ako bolo spomenuté v časti 5.2, pre implementáciu aplikácie bežiacej na webovom serveri bol využitý framework ASP.NET Core. Ten využíva návrhový vzor model-pohľad-kontrolér (angl. Model-View-Controller, MVC), ktorý rozdeľuje zodpovednosti a umožňuje tak lepšie oddeliť dátové a prezentačné časti. Model reprezentuje stav aplikácie a združuje operácie, ktoré vykonáva webový server. Pohľad predstavuje užívateľské rozhranie, teda samotnú webovú stránku. Kontrolér je potom zodpovedný za spracovanie užívateľských požiadaviek, zmenu modelu a výber príslušného pohľadu, ktorý sa má zobraziť. Taktiež bolo spomenuté, že okrem poskytovania webových stránok podporuje tento server aj prijímanie požiadaviek na vyhľadávanie pomocou API. Rozdiel je iba v tom, že pohľad v tomto prípade už nie je webová stránka, ale odpoveď vo formáte JSON.

```
1 [HttpGet("Progress/{id}")]
2 public SearchProgress Progress(long id, [FromServices] IBusProvider busProvider)
```

Ukážka 6.3: Predpis API funkcie obsluhujúcej požiadavky na priebeh vyhľadávania

Za zobrazenie príslušnej stránky je v projekte `WebApp` zodpovedný kontrolér `AppController`. API je rozdelené medzi dva kontroléry. Požiadavky týkajúce sa priamo vyhľadávania spracováva `SearchController`, zatiaľ čo požiadavky na dáta o molekulách preberá `MoleculeController`. K výberu kontroléra a metódy zodpovednej za spracovanie požiadavky dochádza na základe URL adresy. Východzí formát použitý pre mapovanie URL je `"{controller}/{action}/{id?}"`. V prípade zaslania požiadavky metódou GET na URL `/api/Search/Progress/2` by tak došlo k vyvolaniu metódy `Progress` v kontroléri `SearchController`, ktorej predpis je možné vidieť v ukážke 6.3. Metóde by bol potom predaný argument 2, ktorý je v URL substituovaný za `{id}`, ako hodnota prvého parametru. Druhý parameter sa získa pomocou mechanizmu vkladania závislostí (angl. Dependency Injection), ktorý umožňuje namiesto explicitného uvedenia argumentu jeho vyhľadanie v zaregistrovaných službách. `IBusProvider` deklaruje rozhranie potrebné pre komunikáciu s aplikačnými servermi pomocou RabbitMQ. Toto rozhranie implementuje trieda `BusProvider`, ktorej inštancia je zaregistrovaná ako služba pomocou volania `services.AddSingleton<IBusProvider>(new BusProvider())`. Zaregistrovanie inštancie ako singleton umožňuje jej zdieľanie naprieč všetkými požiadavkami. Pri vytvorení inštancie triedy `BusProvider` dôjde podobne ako v prípade aplikačného servera k vytvoreniu objektu `IBus`, ktorý predstavuje spojenie s RabbitMQ serverom. Po prijatí požiadavky od klienta cez API je volaním metódy `Request` na objekt `IBus` táto požiadavka predaná aplikačnému serveru vložením do príslušnej RabbitMQ fronty.

Pohľady sú vytvorené v HTML s využitím spomínanej syntaxe Razor, ktorá v tomto prípade zabezpečuje predanie dát a vygenerovanie správnych URL v JavaScript kóde. Pohľad úvodnej stránky je tvorený iba štruktúrnym editorom a elementom pre výber typu vyhľadávania. Po potvrdení vstupov je užívateľ presmerovaný na pohľad prezentujúci výsledky, kde je najskôr uvedený priebeh vyhľadávania a po jeho dokončení sú zobrazené výsledky formou tabuľky s náhľadom na danú molekulu, odkazom do systému PubChem a InChI identifikátorom. Tieto dáta sú získané asynchrónne opäť pomocou API. Na jednej výsledkovej stránke sa zobrazuje najviac 20 výsledkov vyhľadávania. K rozdeleniu výsledkov na stránky bola použitá knižnica jQuery Pagination[1].

[1] Odkaz na stiahnutie: http://esimakin.github.io/twbs-pagination/

6.6 Jednotkové testy

Pre overenie správnosti kritických častí systému boli vytvorené jednotkové testy (angl. unit tests). Jedná sa o kód, ktorého úlohou je testovať funkčnosť istej malej časti systému, napríklad jednej metódy alebo triedy. V práci boli použité pre otestovanie správneho generovania InChI a InChIKey a taktiež pre overenie výsledkov algoritmov hľadajúcich izomorfný podgraf. V prvom prípade bolo vytvorenie testovacej sady jednoduché, keďže každý záznam v súbore SDF stiahnutom zo systému PubChem obsahoval správny InChI reťazec. Test tak pozostáva z načítania záznamu do dátovej štruktúry molekuly, vygenerovania InChI a porovnania s uloženým reťazcom. V druhom prípade bola testovacia množina vytvorená algoritmom v projekte `UtilityConsoleApp` a triede `UnitTestsHelpers`. Ten umožňuje vytvoriť testovaciu sadu pozostávajúcu zo vzorov, kandidátov a údaju, či sa vzor v kandidátnej molekule nachádza. Pre toto overenie slúži ako referenčná implementácia algoritmus VF2 z knižnice RDKit. Vzor je súčasťou testovacej sady, ak počet kandidátov skutočne obsahujúcich vzor presiahne určitú hranicu. To zabezpečí, že sa jednak overí nájdenie výskytu, ale aj potvrdenie absencie podgrafu v štruktúre. Predtým, než je možné vytvoriť testovaciu sadu, je nutné naplniť databázu aspoň miliónom záznamov.

```
1  [TestFixture]
2  public class ArrayTests
3  {
4     [Test]
5     public void Test123()
6     {
7        int[] array = new int[] { 1, 2, 3 };
8        Assert.That(array, Has.Exactly(1).EqualTo(3));
9     }
10 }
```

Ukážka 6.4: Príklad jednotkového testu pre overenie obsahu poľa

Pre vytváranie testov a ich spúšťanie bola zvolená knižnica NUnit[1]. Trieda obsahujúca testy sa označí atribútom `TestFixture` a jednotlivé metódy predstavujúce testy atribútom `Test`. Ukážka 6.4 je príklad jednoduchého jednotkového testu. Pomocou metódy `Assert.Fail` je možné v prípade chyby označiť test ako neúspešný a pripojiť aj chybovú hlášku. Pre jednotkové testy bol vytvorený samostatný projekt `UnitTests`, ktorý obsahuje dva testy pre generovanie InChI, dva pre rekonštrukciu molekuly z InChI a pre každý z implementovaných algoritmov pre hľadanie izomorfného podgrafu dva testy, z ktorých jeden rýchlo overí funkčnosť metódy a druhý otestuje algoritmus dôkladne využitím pripravenej testovacej sady.

[1] Repozitár: https://github.com/nunit/nunit

Kapitola 7

Merania a diskusia výsledkov

V tejto kapitole sú zhrnuté merania, na základe ktorých bolo rozhodované pri návrhu systému, a taktiež merania z výsledného systému s reálnymi dátami. Súčasťou každého merania je diskusia, v ktorej sú dosiahnuté výsledky uvedené do širšieho kontextu. Všetky merania boli vykonané na počítači s touto špecifikáciou:

- CPU – AMD Ryzen 5 1600
- RAM – 8GB, DDR4, 2933Mhz
- HDD – WD5000AAKS, 500GB, 7200 otáčok/s
- SSD – Samsung 850 EVO, 500GB

Pri meraniach akýchkoľvek diskových operácií je vhodné vypnúť antivírusový softvér, pretože môže dochádzať ku kontrole pri otvorení súboru, čo by výrazne skreslilo namerané hodnoty. Medzi meraniami, ktoré reálne čítajú obsah súboru, je navyše potrebné vyčistiť cache pamäť súborového systému. K tomu bol použitý nástroj RAMMap[1], v ktorom je túto funkciu možné nájsť v menu pod voľbou `Empty->Empty Standby List`. Väčšina týchto meraní je implementovaná triedami `StorageExperiments` a `SubgraphExperiments` v projekte `UtilityConsoleApp`. Projekt je však nutné manuálne pozmeniť pre vykonanie požadovaných meraní.

7.1 Merania algoritmov pre hľadanie izomorfného podgrafu

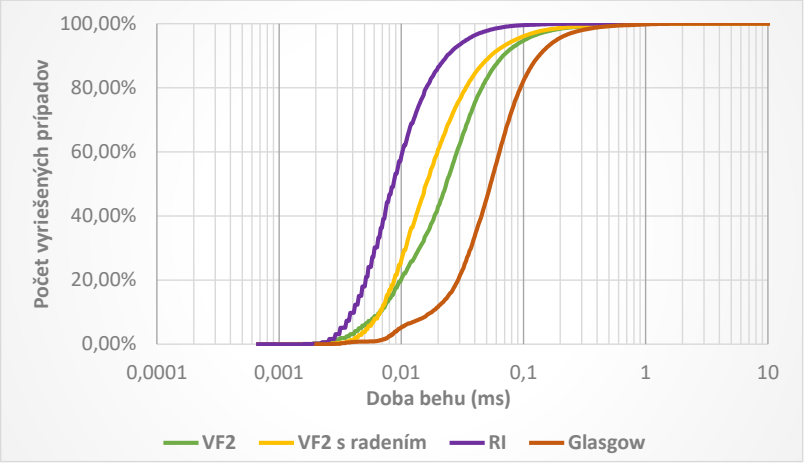
V tejto časti sú predstavené výsledky meraní a porovnanie algoritmov pre hľadanie izomorfného podgrafu a ich variánt. Konkrétne sa jedná o algoritmy VF2, VF2 s radením, Glasgow, RI a teoretický algoritmus Best, ktorý pre konkrétnu dvojicu grafov zvolí vždy najrýchlejší algoritmus spomedzi dostupných. Radenie pri VF2 spočíva v zoradení prehľadávaných vrcholov podľa protónového čísla prvku a stupňa vrcholu. Menej bežné prvky s veľkým počtom susedov sa tak navštívia ako prvé. Meranie bolo vykonané pre dva prípady použitia. V prvom sa jednalo o vyhľadávanie v takzvaných fragmentoch, čo sú molekuly získané výpočtom zo zadanej molekuly pomocou operácie fragmentácie. Pre chemických pracovníkov má zmysel podštruktúrne vyhľadávať v takto nadobudnutej množine molekúl a je tu vysoká pravdepodobnosť nájdenia podštruktúry. V druhom prípade prebiehalo meranie nad reálnou databázou naplnenou miliónom chemických látok získaných z databázy PubChem. Ako vzory bolo použitých tisíc najčastejších podštruktúr z rovnakej databázy. Tie predstavujú

[1] Odkaz na stiahnutie: https://docs.microsoft.com/en-us/sysinternals/downloads/rammap

najhorší možný prípad pri vyhľadávaní, pretože budú nájdené vo veľkom množstve záznamov. Zoznam týchto podštruktúr bol získaný kontaktovaním autora článku [13], ktorý sa zaoberal vizuálnym rozmiestnením štruktúr v 2D priestore. S cieľom skrátiť trvanie merania boli zo zoznamu odstránené príliš všeobecné vzory obsahujúce menej ako osem atómov. Vo výsledku sa teda jednalo o 902 vzorov. Aby merania zodpovedali reálnemu prípadu použitia, algoritmus bol spustený iba pre databázové záznamy, ktoré vyhoveli testom odtlačkov. Pre každú dvojicu grafov bolo meranie zopakované päťkrát a z nameraných časov bol zvolený medián, aby prípadné vyťaženie procesora iným procesom nemalo na meranie vplyv. Priemerná hodnota by v prípade namerania výnimočne dlhého času bola skreslená. Následne bola pre každý algoritmus vypočítaná priemerná doba behu a počet prípadov, kedy bol daný algoritmus najrýchlejší spomedzi všetkých meraných. V prípade fragmentov bol celkový počet inštancií 41882 a vzor sa našiel v 4441 prípadoch. Pre databázu boli tieto čísla 6485939 inštancií a 823329 prípadov nájdenia vzoru.

Algoritmus	Fragmenty		Databáza	
	Doba behu (µs)	Najrýchlejší (počet)	Doba behu (µs)	Najrýchlejší (počet)
VF2	8.617	15476	36.284	720534
VF2 s radením	11.44	1611	28.887	88661
Glasgow	56.609	0	75.112	200388
RI	6.313	24795	12.728	5476356
Best	5.453		11.976	

Tabuľka 7.1: Porovnanie algoritmov pre hľadanie izomorfného podgrafu. Doba behu je priemerná hodnota pre jednu inštanciu a stĺpec najrýchlejší uvádza, koľko inštancií vyriešil daný algoritmus v najkratšom čase.



Obr. 7.1: Porovnanie algoritmov pre hľadanie izomorfného podgrafu formou kumulatívneho histogramu. Interpretovať ho je možné ako počet inštancií, pre ktoré doba behu neprekračuje určitý čas. Napríklad v prípade RI neprekračuje doba behu 10 µs v 60 % všetkých inštancií.

Výsledky merania sú uvedené v tabuľke 7.1. Obrázok 7.1 zobrazuje namerané časy pre databázu formou kumulatívneho histogramu. Ako je možné vidieť, RI je najrýchlejší al-

goritmus v oboch prípadoch a za teoretickým algoritmom zaostáva v prípade databázového merania iba o necelých 5 %. Naopak algoritmus Glasgow sa pre vyhľadávanie v pomerne malých grafoch ukázal ako veľmi nevhodný. Jednak za to môže dlhá inicializácia domén a premenných, ktorá u ostatných algoritmov chýba, ale tiež princíp vytvárania konfliktných množín sa zdá byť užitočný až pri grafoch s veľkým počtom vrcholov a hrán. Radenie vrcholov sa pri algoritme VF2 ukázalo byť prospešné pre vyhľadávanie v databáze, no pri prehľadávaní fragmentov viedlo k spomaleniu. Na základe týchto výsledkov nemá zmysel algoritmy akýmkoľvek spôsobom kombinovať alebo dynamicky voliť a pre všetky podštruktúrne vyhľadávania sa použije algoritmus RI.

7.2 Merania úložiska štruktúrnych dát

V tejto časti sú uvedené výsledky meraní, na základe ktorých bolo možné informovane zvoliť vhodnejší spôsob uloženia štruktúrnych dát. Najdôležitejším meraným parametrom bola doba získania požadovaných dát z úložiska.

7.2.1 Prístupová doba v súborovom systéme

V prípade súborového systému bolo nutné najskôr určiť, do ako veľkých zložiek je vhodné dáta rozdeliť, keďže sa predpokladá, že uloženie niekoľkých miliónov súborov do jednej zložky by mohlo mať vplyv na rýchlosť otvorenia súboru. Okrem prístupovej doby k súboru v rôzne veľkých zložkách bola táto doba meraná aj pre stromovú štruktúru s dvomi úrovňami. Každá zložka v strome obsahovala 100 ďalších zložiek alebo na najnižšej úrovni súborov. Merania boli vykonané pre platňový disk aj SSD. Otvorených bolo 50 náhodných súborov v zložke a meranie zopakované stokrát. Výsledná doba v tabuľke 7.2 je priemerom všetkých meraní. Variačný koeficient, čiže podiel smerodajnej odchýlky a priemeru, sa pri všetkých meraniach pohyboval okolo hodnoty 0.3.

Počet	100	500	1000	5000
HDD	97.478	103.041	106.291	117.142
SSD	102.626	113.461	129.938	175.376
Počet	10000	100000	300000	Strom (100)
HDD	120.725	1059.978	3485.66	96.07
SSD	214.802	455.916	529.47	110.796

Tabuľka 7.2: Prístupová doba v µs k súboru v závislosti na počte súborov v zložke pre súborový systém NTFS.

Výsledky potvrdzujú, že prístupová doba závisí od veľkosti zložky, nie však od zanorenia súboru v stromovej štruktúre. Ideálna veľkosť zložky sa pohybuje okolo hodnoty 1000. Platňový disk umožňuje aj väčšie zložky, ale nad 10000 súborov prístupová doba drasticky narastá. V prípade SSD disku je tento nárast pozvoľnejší.

7.2.2 Porovnanie súborového systému a MongoDB

Po určení vhodnej veľkosti zložiek bolo možné vytvoriť úložisko v súborovom systéme a porovnať rýchlosť získania dát s databázovým systémom MongoDB. V súborovom systéme

na SSD disku bola vytvorená dvojúrovňová stromová štruktúra, kde každá zložka obsahovala najviac 1000 súborov. Každá chemická štruktúra bola uložená ako jeden súbor, ktorého obsah sa čítal v prostredí .Net pomocou funkcie `File.ReadAllBytes`. Okrem sekvenčného čítania po jednom súbore bol meraný čas aj pre paralelný prístup. Paralelizácia bola realizovaná pomocou rovnakého mechanizmu TPL Dataflow, ktorý sa využíva pre paralelizáciu podštruktúrneho vyhľadávania. Jednu dávku predstavovalo 100 súborov a maximálna miera paralelizácie bola nastavená na 60.

V databáze MongoDB uloženej taktiež na SSD disku bola vytvorená nová kolekcia záznamov, ku ktorej sa pristupovalo pomocou oficiálneho MongoDB ovládača pre .Net. K získaniu požadovaných záznamov bola použitá funkcia `collection.Find`, ktorej bol ako parameter predaný filter so zoznamom identifikátorov požadovaných záznamov. Keďže tento spôsob umožňuje získať viacero záznamov jedným dopytom, boli zmerané časy pre rôzny počet súbežne získaných záznamov.

	NTFS		MongoDB			
	Sekvenčne	Paralelelne	1	10	100	1000
Doba získania dát (µs)	211.936	37.854	234.376	35.108	13.408	12.137

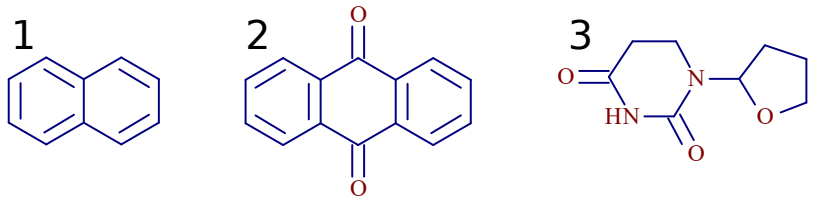
Tabuľka 7.3: Priemerná doba získania serializovaných dát chemickej štruktúry z úložiska

Do oboch dátových úložísk bolo uložených milión chemických štruktúr s priemernou veľkosťou 524B v serializovanej reprezentácii. Následne sa merala doba získania serializovaných dát pre 10000 náhodných chemických štruktúr, každé meranie bolo desaťkrát zopakované a výsledky spriemerované. V tabuľke 7.3 je možné nájsť výslednú priemernú dobu načítania dát jednej chemickej štruktúry pre uvedené metódy.

Ako je možné vidieť vo výsledkoch, získanie dát pre jeden záznam databázy približne zodpovedá sekvenčnému získaniu dát zo súboru, avšak pri vyžiadaní viacerých záznamov z databázy v rámci jedného dopytu je databázové riešenie rádovo rýchlejšie. Oproti paralelnému načítaniu dát zo súboru nie je zrýchlenie až tak výrazné, ale je stále niekoľkonásobné. Najväčší rozdiel v dobe získania záznamov z databázy je medzi získaním jedného a desiatich záznamov súbežne. S ďalej sa zvyšujúcim počtom je postupne toto zrýchlenie menej výrazné, zatiaľ čo rastú pamäťové nároky na uloženie dát. Ako optimálna sa preto javí hodnota medzi 100 až 1000 súbežne získanými záznamami. Keďže je úložisko využívajúce databázový systém rýchlejšie, ľahšie škálovateľné vďaka operácii trieštenia a je ho možné sprístupniť cez sieť viacerým aplikačným serverom súčasne, bolo nakoniec vo výslednom systéme uprednostnené pred súborovým systémom.

7.3 Všeobecné meranie výkonnosti systému

V tejto časti sú prezentované výsledky meraní nad výsledným systémom. V databáze bolo celkovo uložených 9634487 záznamov získaných z databázy PubChem a pomocou webového rozhrania boli v databáze vyhľadávané tri vzory, ktoré je možné vidieť na obrázku 7.2. Vzory boli zvolené tak, aby ukázali rôzne prípady, ktoré môžu počas vyhľadávania nastať. Tieto prípady budú bližšie objasnené v nasledujúcich častiach pre každý typ vyhľadávania zvlášť.



Obr. 7.2: Trojica vzorov, ktoré slúžili pre zhodnotenie výkonnosti výsledného systému

7.3.1 Identické vyhľadávanie vo výslednom systéme

Pri identickom vyhľadávaní je zaujímavé okrem celkovej doby operácie merať aj dve najkritickejšie zložky. Prvou je vygenerovanie InChI a InChIKey pre vstupnú molekulu. Tou druhou je samotné vyhľadanie a získanie výsledku z databázy. Prvé dva vzory z obrázku 7.2 je možné v databáze nájsť. Pre tretí neexistuje príslušný záznam, vďaka čomu je známa doba prehľadania celej databázy. Vyhľadávanie bolo pre každý vzor spustené päťkrát a namerané časy spriemerované.

	Vzor		
	1	2	3
Generovanie InChI (ms)	0.417	0.477	0.502
Vyhľadanie v DB (ms)	1.588	1.597	1.613
Celkový čas (ms)	2.526	2.174	2.204

Tabuľka 7.4: Priemerná doba operácií vykonávaných pri identickom vyhľadávaní

Výsledky meraní je možné nájsť v tabuľke 7.4. Na ich základe je možné povedať, že podstatnú časť celkovej doby trvá práve vyhľadávanie v databáze, no vygenerovanie InChIKey nie je zanedbateľnou operáciou. Vďaka využitiu indexu pre InChIKey je získanie príslušného záznamu veľmi rýchle. S vyšším počtom záznamov v databáze je možné očakávať, že sa mierne zvýši doba potrebná pre vykonanie tejto operácie. Vzhľadom na metódu indexovania, konkrétne pomocou B stromu, by mal byť tento nárast logaritmický. Pre identické vyhľadávanie bola v časti 4.3 určená maximálna doba vyhľadávania 10 ms. Túto požiadavku sa teda podarilo splniť.

7.3.2 Podštruktúrne vyhľadávanie vo výslednom systéme

Vzhľadom na podštruktúrne vyhľadávanie sú zvolené vzory odstupňované podľa ich špecifickosti. Prvý vzor je veľmi všeobecný a má veľkú množinu kandidátov tvorenú 393676 záznamami a vzor v skutočnosti obsahuje ako podgraf 266821 z nich. Druhý vzor sa dá považovať za bežný prípad, kedy počet kandidátov je 10634 a výskyt vzoru je možné nájsť v 1703 záznamoch. Posledný vzor má extrémne malú množinu kandidátov. V databáze existuje pre tento vzor iba 273 kandidátov a 234 z nich vzor obsahuje. Vyhľadávanie bolo spustené pre každý vzor päťkrát a časy v tabuľke 7.5 zodpovedajú priemernej dobe vykonania danej operácie. Okrem celkového času bola meraná napríklad doba získania kandidátov z databázy obsahujúcej odtlačky a čas, ktorý bol potrebný pre overenie výskytu vzoru v kandidátoch. Druhá operácia sa deje paralelne po dávkach s určitou veľkosťou, preto nebolo možné jednoducho zmerať, koľko trvá celkovo získanie štruktúrnych dát z databázy a

koľko ich následné prehľadanie algoritmom. Namiesto toho sú v tabuľke uvedené priemerné hodnoty pre tieto operácie vzhľadom na jednu dávku.

	Vzor		
	1	2	3
Vytvorenie odtlačku pre vzor (ms)	1.291	3.037	1.365
Získanie kandidátov z DB (s)	14.631	14.277	14.394
Získanie štruktúrnych dát (ms/dávka)	1078.04	152.984	11.527
Deserializácia a beh algoritmu (ms/dávka)	1853.681	899.645	26.121
Skontrolovanie kandidátov (s)	34.8	1.144	0.038
Celkový čas (s)	49.5	15.425	14.433

Tabuľka 7.5: Priemerná doba operácií vykonávaných pri podštruktúrnom vyhľadávaní

Z výsledkov je zrejmé, že doba vyhľadávania závisí predovšetkým od počtu kandidátov. V prípade prvého vzoru, pre ktorý je množina kandidátov veľká, trvá najdlhšie skontrolovanie výskytu vzoru. Priemerná doba deserializácie a behu algoritmu je podľa očakávaní dlhšia než doba získania štruktúrnych dát. S ohľadom na merania z tabuliek 6.2, 7.1 a 7.3 by však rozdiel medzi týmito hodnotami mal byť výraznejší. Sčítaním doby deserializácie formátu MPM a priemernej doby behu algoritmu RI získavame totiž pre jednu dvojicu $64.91 + 12.728 = 77.638$µs, zatiaľ čo očakávaná doba získania štruktúrnych dát pre jeden záznam je iba 12.137 µs. V prípade druhého vzoru sú namerané hodnoty viac podľa očakávaní. Dôkladnejšou analýzou tohto javu sa ako zdroj problému javí nedostatok pamäte RAM pre MongoDB server, ktorého výkonnosť klesá pri obsluhe veľkého množstva požiadaviek a nedostatku výpočtových prostriedkov. 8 GB pamäti by malo byť postačujúcich, no v prípade spustenia všetkých súčastí systému na jednom počítači je pre MongoDB dostupná iba malá časť z uvedenej hardvérovej špecifikácie.

Pre menšie množiny kandidátov sa do popredia dostáva samotná operácia získania kandidátov. Trvanie tejto operácie je v meraniach viac-menej konštantné, no so zväčšujúcou sa databázou bude narastať lineárne. To je pomerne veľký problém, ktorý by však bolo možné riešiť vytvorením špeciálneho GiST indexu pre odtlačky v PostgreSQL databáze. Princíp indexu je založený na stromovej štruktúre vytvorenej na základe prefixu odtlačku. Tá by umožnila v prípade nájdenia nevyhovujúceho odtlačku, ktorý nemá nastavené správne bity, vynechať všetky záznamy v databáze s rovnakým prefixom, ktoré teda taktiež nemôžu patriť do množiny kandidátov. V priebehu písania práce sa však nepodarilo vytvoriť správnu implementáciu tohto indexu, preto nie je súčasťou výsledného riešenia.

Kapitola 8

Záver

Cieľom tejto diplomovej práce bolo navrhnúť a implementovať systém umožňujúci vyhľadávanie chemických látok na základe ich štruktúry. K tomu bolo nutné sa najskôr oboznámiť s konceptmi a algoritmami využívanými nielen v teórii grafov, ale aj v oblasti chemoinformatiky. Podstatnú časť práce predstavoval výber technológii a skombinovanie existujúcich metód tak, aby zodpovedali vytýčeným požiadavkám.

Výsledkom práce je zdôvodnený návrh a otestovaná implementácia systému podporujúceho operácie identického a podštruktúrneho vyhľadávania. Tento systém tvorí webový server, ktorý poskytuje webové stránky a prijíma požiadavky na vyhľadávanie pomocou API, a aplikačný server, ktorý implementuje algoritmy pre vyhľadávanie a komunikuje s databázami, v ktorých sú uložené štruktúrne dáta a metadáta molekúl. Návrh umožňuje jednoduché škálovanie jednotlivých častí systému v prípade potreby, napríklad vytvorenie viacerých aplikačných serverov alebo využitie trieštenia pre zvýšenie rýchlosti získania štruktúrnych dát z databázy.

Identické vyhľadávanie je riešené pomocou unikátneho identifikátora molekúl InChI a jeho skrátenej reprezentácie InChIKey. Podštruktúrne vyhľadávanie je realizované kombináciou molekulárnych odtlačkov, ktoré umožňujú výrazne zmenšiť množinu prehľadávaných záznamov, a rýchleho algoritmu pre hľadanie izomorfného podgrafu. V práci bolo porovnaných viacero takýchto algoritmov. Z hľadiska hľadania výskytu vzoru v chemických štruktúrach bol najvhodnejší algoritmus RI.

Meraniami bolo overené, že požiadavky na systém týkajúce sa maximálneho času trvania jednotlivých operácií boli splnené. Okrem toho boli dodržané aj ďalšie požiadavky, týkajúce sa napríklad vytvorenia optimálneho formátu pre ukladanie štruktúrnych dát v rámci systému alebo obmedzení na prípustné technológie. Taktiež bolo úplne splnené zadanie diplomovej práce. Grafové databázy sa ukázali ako nie príliš vhodné pre riešenie daného problému, no bola predstavená jedna možnosť ich použitia, ktorá by v závislosti na spôsobe využívania systému mohla viesť k ďalšiemu zrýchleniu operácie podštruktúrneho vyhľadávania.

Ďalší vývoj práce spočíva v reálnom nasadení systému v produkčnom prostredí. Okrem toho je nutné dokončiť databázový index pre podštruktúrne odtlačky, ktorý sa v rámci práce nepodarilo implementovať a ktorý je dôležitý pre rýchle získanie množiny kandidátov v rámci podštruktúrneho vyhľadávania vo veľkých databázach. Ďalšie rozšírenie systému by mohla predstavovať podpora ďalších operácii, napríklad rýchleho vyhľadávania na základe podobnosti, k čomu by bolo možné použiť napríklad kruhové odtlačky.

Literatúra

[1] BIOVIA: CTfile Formats. [cit. 2018-04-27].
URL http://accelrys.com/products/collaborative-science/biovia-draw/ctfile-no-fee.html

[2] Bonnici, V.; Giugno, R.; Pulvirenti, A.; aj.: A subgraph isomorphism algorithm and its application to biochemical data. *BMC bioinformatics*, ročník 14, č. 7, 2013: str. S13.

[3] Boussemart, F.; Hemery, F.; Lecoutre, C.; aj.: Boosting systematic search by weighting constraints. In *Proceedings of the 16th European Conference on Artificial Intelligence*, IOS Press, 2004, s. 146–150.

[4] Carhart, R. E.; Smith, D. H.; Venkataraghavan, R.: Atom pairs as molecular features in structure-activity studies: definition and applications. *Journal of Chemical Information and Computer Sciences*, ročník 25, č. 2, 1985: s. 64–73.

[5] Cordella, L. P.; Foggia, P.; Sansone, C.; aj.: Performance evaluation of the VF graph matching algorithm. In *Image Analysis and Processing, 1999. Proceedings. International Conference on*, IEEE, 1999, s. 1172–1177.

[6] Cordella, L. P.; Foggia, P.; Sansone, C.; aj.: A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence*, ročník 26, č. 10, 2004: s. 1367–1372.

[7] Dalke, A.: Implementing the CACTVS/PubChem substructure keys. 2011, [cit. 2018-03-17].
URL http://www.dalkescientific.com/writings/diary/archive/2011/01/20/implementing_cactvs_keys.html

[8] Daylight Chemical Information Systems: Fingerprints - Screening and Similarity. 2008, [cit. 2018-04-01].
URL http://www.daylight.com/dayhtml/doc/theory/theory.finger.html

[9] Daylight Chemical Information Systems: SMARTS – A Language for Describing Molecular Patterns. 2008, [cit. 2018-02-01].
URL http://www.daylight.com/dayhtml/doc/theory/theory.smarts.html

[10] Daylight Chemical Information Systems: SMILES – A Simplified Chemical Language. 2008, [cit. 2018-02-01].
URL http://www.daylight.com/dayhtml/doc/theory/theory.smiles.html

[11] Demel, J.: *Grafy a jejich aplikace*. Academia, 2002, ISBN 80-200-0990-6.

[12] Ehrlich, H.-C.; Rarey, M.: Systematic benchmark of substructure search in molecular graphs-From Ullmann to VF2. *Journal of cheminformatics*, ročník 4, č. 1, 2012: str. 13.

[13] Ertl, P.; Rohde, B.: The molecule cloud-compact visualization of large collections of molecules. *Journal of cheminformatics*, ročník 4, č. 1, 2012: str. 12.

[14] Fielding, R. T.; Taylor, R. N.: *Architectural styles and the design of network-based software architectures*, ročník 7. University of California, Irvine Doctoral dissertation, 2000.

[15] Flowers, P.; Theopold, K.; Langley, R.; aj.: *Chemistry*. OpenStax, Rice University, 2016, ISBN 9781938168390.

[16] Heller, S. R.; McNaught, A.; Pletnev, I.; aj.: InChI, the IUPAC international chemical identifier. *Journal of cheminformatics*, ročník 7, č. 1, 2015: str. 23.

[17] Klein, D.: *Organic Chemistry, 2nd Edition:*. Wiley, 2013, ISBN 9781118795705.

[18] Kotthoff, L.; Mccreesh, C.; Solnon, C.: Portfolios of subgraph isomorphism algorithms. In *International Conference on Learning and Intelligent Optimization*, Springer, 2016, s. 107–122.

[19] Landrum, G.: Fingerprint-based substructure screening 1. 2013, [cit. 2018-03-17]. URL http://rdkit.blogspot.sk/2013/11/fingerprint-based-substructure.html

[20] McCreesh, C.; Prosser, P.: A parallel, backjumping subgraph isomorphism algorithm using supplemental graphs. In *International Conference on Principles and Practice of Constraint Programming*, Springer, 2015, s. 295–312.

[21] Neo4j Staff: Webinar Follow Up: Intro to Graph Databases. 2012, [cit. 2018-01-12]. URL https://neo4j.com/blog/webinar-follow-up-intro-to-graph-databases/

[22] OEChem: 12.7 Smallest Set of Smallest Rings (SSSR) considered Harmful. 2008, [cit. 2018-04-14].
URL https://www.ics.uci.edu/~dock/manuals/oechem/pyprog/node123.html

[23] O'Boyle, N. M.; Sayle, R. A.: Comparing structural fingerprints using a literature-based similarity benchmark. *Journal of cheminformatics*, ročník 8, č. 1, 2016: str. 36.

[24] Piper, A.: Choosing Your Messaging Protocol: AMQP, MQTT, or STOMP. 2013, [cit. 2018-04-15].
URL https://blogs.vmware.com/vfabric/2013/02/choosing-your-messaging-protocol-amqp-mqtt-or-stomp.html

[25] Plesník, J.: *Grafové algoritmy*. Veda, 1983.

[26] Pletnev, I.; Erin, A.; McNaught, A.; aj.: InChIKey collision resistance: an experimental testing. *Journal of cheminformatics*, ročník 4, č. 1, 2012: str. 39.

[27] PostgreSQL Global Development Group: Extending SQL – C-Language Functions. 2018, [cit. 2018-04-28].
URL https://www.postgresql.org/docs/10/static/xfunc-c.html

[28] Poulson, T.; Walter, L.: *Introduction to Chemistry*. CreateSpace Independent Publishing Platform, 2012, ISBN 9781478298601.

[29] PubChem: PubChem Substructure Fingerprint. 2009, [cit. 2018-02-14].
URL ftp://ftp.ncbi.nlm.nih.gov/pubchem/specifications/pubchem_fingerprints.txt

[30] PubChem: Data Sources. 2018, [cit. 2018-04-13].
URL https://pubchem.ncbi.nlm.nih.gov/sources

[31] PubChem: Programmatic Access. 2018, [cit. 2018-04-13].
URL http://pubchemdocs.ncbi.nlm.nih.gov/programmatic-access

[32] Rathle, P.: Official Release: 3 Essentials of Neo4j 3.0, from Scale to Productivity & Deployment. 2016, [cit. 2018-01-13].
URL
https://neo4j.com/blog/neo4j-3-0-massive-scale-developer-productivity

[33] Robinson, I.; Webber, J.; Webber, J.; aj.: *Graph Databases*. O'Reilly, 2013, ISBN 9781449356262.

[34] Rogers, D.; Hahn, M.: Extended-connectivity fingerprints. *Journal of chemical information and modeling*, ročník 50, č. 5, 2010: s. 742–754.

[35] Skonnard, A.: Understanding SOAP. 2003, [cit. 2018-04-15].
URL https://msdn.microsoft.com/en-us/library/ms995800.aspx

[36] solid IT: DB-Engines Ranking. 2018, [cit. 2018-01-06].
URL https://db-engines.com/en/ranking

[37] Stein, S. E.; Heller, S. R.; Tchekhovskoi, D. V.: The IUPAC Chemical Identifier – Technical Manual. *National Institute of Standards and Technology, Gaithersburg, Maryland, US*, 2017.

[38] Neo Technology: The Definitive Guide to Graph Databases. 2016, [cit. 2018-01-06].
URL https://neo4j.com/resources/rdbms-developer-graph-white-paper/

[39] Ullmann, J. R.: An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, ročník 23, č. 1, 1976: s. 31–42.

[40] Ullmann, J. R.: Bit-vector algorithms for binary constraint satisfaction and subgraph isomorphism. *Journal of Experimental Algorithmics (JEA)*, ročník 15, 2010: s. 1–64.

Príloha A

Obsah priloženého DVD

Na zadnej obálke práce je možné nájsť DVD s nasledujúcim obsahom:

- **Bin/** zložka obsahujúca samorozbaľovací archív s aplikáciami skompilovanými pre systém Windows. Súčasťou archívu je aplikačný server, publikovaná webová aplikácia a nástroj ManagementTool, ktorý je možné použiť pre vloženie záznamov z SDF súborov do databázy. Zložka taktiež obsahuje inštalačné súbory pre potrebné programy.
- **Doc/** zložka obsahujúca dokumentáciu zdrojových kódov vo forme HTML stránok vytvorenú pomocou nástroja Doxygen.
- **Latex/** zložka obsahujúca LaTeX zdrojové kódy slúžiace k vygenerovaniu tejto diplomovej práce.
- **Source/** zložka obsahujúca všetky zdrojové kódy. Súčasťou je súbor `README.txt`, ktorý v stručnosti popisuje postup pre spustenie systému.
- **Data/** zložka obsahujúca rôzne dátové súbory:
  - **PubChem/** obsahuje dva súbory SDF, ktoré boli vytvorené filtrovaním milióna PubChem záznamov. Vhodné pre počiatočné naplnenie vlastnej databázy.
  - **Scaffolds/** obsahuje SMILES definície a príslušný SDF súbor s tisíc najčastejšími PubChem podštruktúrami. Tieto dáta poskytol autor článku [13].
  - **Fragments/** obsahuje niekoľko SDF súborov obsahujúcich fragmentácie rôznych molekúl.

Príloha B

Inštrukcie pre spustenie

Vzhľadom na komplexnosť systému je jeho spustenie pomerne náročné. Aby sa tento proces zjednodušil, bolo vytvorených niekoľko skriptov pre systém Windows, ktoré je možné nájsť v zložke `Source/Utilities` a budú používané v nasledujúcich častiach. V prvom rade je však nutné nainštalovať potrebné nástroje a programy, na ktorých je systém postavený. Ich zoznam je nasledovný:

- Visual Studio 2015 alebo novšie – pre skompilovanie zdrojových súborov.
- .NET Core nástroje[1] – pre publikovanie aplikácie webového serveru.
- IIS Express 10[2] – server pre beh webovej aplikácie.
- PostgreSQL 10 – databáza pre metadáta.
- MongoDB 3.6 – databáza pre štruktúrne dáta.
- RabbitMQ 3.7 – zabezpečuje komunikáciu medzi servermi.
- Erlang 20 – potrebný pre RabbitMQ.

Po nainštalovaní nástrojov je nutné upraviť konfiguračný skript `config.cmd` v spomínanej zložke tak, aby hodnoty premenných zodpovedali cestám, do ktorých boli programy nainštalované. Aktuálny obsah súboru je možné použiť ako nápovedu. Navyše je nutné nakonfigurovať projekt `MoleculeSearchDbExtension`, ktorý potrebuje poznať správnu cestu k PostgreSQL, aby mohol pri kompilácii využiť potrebné knižnice. Po otvorení projektu vo Visual Studio je nutné v hornom menu zvoliť `View->Other Windows->Property Manager`. V otvorenom paneli bude položka pre tento projekt, ktorá po rozbalení obsahuje viacero konfigurácií pre rôzne architektúry. Jednu z nich je potrebné otvoriť a zvoliť `PropertySheet`. V novom okne je pod `Common Properties` možné nájsť položku `User Macros`. V nej je potrebné zmeniť hodnotu pre záznam `POSTGRE_FOLDER` na cestu, kam bola nainštalovaná databáza PostgreSQL.

B.1 Kompilácia systému

Zdrojové kódy je možné skompilovať priamo pomocou Visual Studio otvorením súboru `Molecule.sln` a zvolením položky pre túto úlohu v menu `Build->Build Solution`. Keďže je

[1] Odkaz na stiahnutie: https://github.com/dotnet/cli
[2] Odkaz na stiahnutie: https://www.microsoft.com/en-us/download/details.aspx?id=48264

pre každú architektúru pripravená samostatná konfigurácia, bol vytvorený automatizačný skript `batch_build.cmd`, ktorý skompiluje zdrojové kódy pre všetky dostupné konfigurácie. Do procesu kompilácie nebolo možné zahrnúť skopírovanie natívnych knižníc do príslušných zložiek, preto bol pre tento účel vytvorený skript `copy_natives.cmd`, ktorý je nutné po kompilácií spustiť. Tretí skript, `publish_web.cmd`, pripraví webovú aplikáciu, aby mohla byť spustená v rámci serveru IIS Express. Posledným krokom je pridanie modulu do PostgreSQL. K tomu je nutné skopírovať niekoľko súborov priamo do zložiek, ktoré sú súčasťou inštalácie PostgreSQL. Pre zjednodušenie tohto procesu bol vytvorený skript `install_postgre_module.cmd`.

B.2 Spustenie systému

Pred spustením systému je vhodné nakonfigurovať jednotlivé časti, konkrétne webovú aplikáciu, aplikačný server a nástroj ManagementTool. To je možné vytvorením súboru `ConnectionSettings.json` v zložkách obsahujúcich spustiteľné súbory, ktoré sa nachádzajú v zložkách príslušných projektov. Pre webovú aplikáciu je nutné tento súbor vložiť do zložky `Published`. Konfigurácia obsahuje predovšetkým informácie potrebné pre pripojenie k ostatným súčastiam systému ako sú RabbitMQ a databázy. V prípade, že sa konfiguračný súbor nenájde, použijú sa hodnoty uvedené v ukážke B.1, ktoré predpokladajú spustenie všetkých častí systému na rovnakom počítači.

```
{
   "MetadataDatabase": {
      "DataSource": "localhost",
      "UserName": "postgres",
      "Password": "123",
      "DatabaseName": "MoleculeSearch"
   },
   "StorageDatabase": {
      "DataSource": "localhost:27017",
      "DatabaseName": "mol"
   },
   "RabbitMq": {
      "Host": "localhost"
   }
}
```

Ukážka B.1: Príklad konfiguračného súboru `ConnectionSettings.json`

Pre spustenie systému potom existujú dva skripty. Skript `start_services.cmd` spustí časti týkajúce sa backendu, konkrétne RabbitMQ službu, MongoDB proces a dve inštancie aplikačného serveru pomenované server1 a server2. PostgreSQL nie je potrebné spúšťať explicitne, keďže sa vo východzom nastavení spustí pri štarte operačného systému. Následne je možné pomocou skriptu `start_web.cmd` spustiť webovú aplikáciu, ku ktorej sa vytvorí spojenie pomocou predvoleného webového prehliadača.

Pred prvým použitím je potrebné vložiť do systému nejaké dáta. Na priloženom pamäťovom médiu sa v zložke `Data/Pubchem` nachádzajú súbory, ktoré je možné použiť pre tento účel. Nástroj ManagementTool umožňuje vloženie záznamov do databáz. Pre túto operáciu je nutné zvoliť záložku `File Operations` a voľbu `Insert to DB`. Operácia sa spustí pomocou tlačítka `Run` a po skončení bude stav operácie `Done`. Ak boli počas operácie vkladania dát spustené aplikačné servery, je nutné ich reštartovať.